VISIT…



Thinking in C#

Larry O'Brien
and
Bruce Eckel

ISBN 0-13-038572-7
9 780130 385727 94999

Thinking in C#

Larry O'Brien
and
Bruce Eckel



Prentice Hall
Upper Saddle River, New Jersey 07458
www.phptr.com

Dedication

For Tina, who makes me happy — LOB

Overview

Preface: Computer Language	1
1: Those Who Can, Code	1
2: Introduction to Objects	19
3: Hello, Objects	49
4: Controlling Program Flow	87
5: Initialization and Cleanup	149
6: Hiding the Implementation	199
7: Reusing Classes	219
8: Interfaces and Implementation	261
9: Coupling and Cohesion	315
10: Collecting Your Objects	349
11: Error Handling with Exceptions	439
12: I/O in C#	473
13: Reflection and Attributes	511
14: Programming Windows Forms	547
15: GDI+ Overview	659
16: Multithreaded Programming	711
17: XML	773
18: Web Programming	824
A: C# for Visual Basic Programmers	879
B: C# for Java Programmers	883
C: Test-First Programming with NUnit	887

D: Programming the Pocket PC 895
E: C# programming guidelines 903
F: Resources 915
Concordance 917
Class, Method, Property Cross-Reference 921
Index 929

What's Inside

Preface: Computer Language

Prerequisites ... 2
Learning C# ... 3
Goals ... 4
Online documentation ... 5
Exercises ... 5
Source code ... 5
Coding standards ... 7
C# and .NET versions ... 8
Seminars and mentoring ... 8
Errors ... 8
Note on the cover design ... 8
Acknowledgments ... 9

1: Those Who Can, Code

Software economics ... 3
C# and the Internet ... 4
- Static sites ... 5
- Dynamic sites ... 5
- Peer-to-peer ... 5
- Web services ... 6

Security ... 7
Analysis, design, and Extreme Programming ... 7
Strategies for transition ... 9
Management obstacles ... 10
- Return on investment ... 11

Summary ... 17

2: Introduction to Objects

The progress of abstraction ... 19
An object has an interface ... 22
The hidden implementation ... 24
Reusing the implementation ... 26
Inheritance: Reusing the interface ... 27
- Is-a vs. is-like-a relationships ... 31

Interchangeable objects with polymorphism ... 32
- Abstract base classes and interfaces ... 37

Object landscapes and lifetimes ... 37
- Collections and iterators ... 38
- The singly rooted hierarchy ... 40
- Collection libraries and support for easy collection use ... 41
- The housekeeping dilemma: who should clean up? ... 42

Exception handling: dealing with errors ... 43
Multithreading ... 44
Persistence ... 45
Remote objects ... 46
Summary ... 46

3: Hello, Objects

You manipulate objects with references ... 49
You must create all the objects ... 50
- Where storage lives ... 50
- Arrays in C# ... 52
- Special case: value types ... 52

You never need to destroy an object ... 53
- Scoping ... 53
- Scope of objects ... 54

Creating new data types: class ... 55
- Fields, properties, and methods ... 55

Methods, arguments, and return values ... 58
- The argument list ... 59

Attributes and meta-behavior ... 60

Delegates....61
Properties....62
Creating new value types....63
Enumerations....64
Structs....64
Building a C# program....65
Name visibility....67
Using other components....69
The **static** keyword....70
Putting it all together....71
Compiling and running....73
Fine-tuning compilation....75
The Common Language Runtime....75
Comments and embedded documentation....79
Documentation Comments....80
Documentation example....82
Coding style....84
Summary....85
Exercises....85
4: Controlling Program Flow....87
Using C# operators....87
Precedence....87
Assignment....88
Aliasing during method calls....89
Aliasing and object state....90
Aliasing and the **ref** keyword....92
Beyond aliasing with **out**....95
Mathematical operators....98
Unary minus and plus operators....100
Auto increment and decrement....100
Relational operators....101
Testing object equivalence....102
Logical operators....103
Short-circuiting....104
Bitwise operators....105
Shift operators....106
Ternary if-else operator....111
The comma operator....112
Common pitfalls when using operators....113
Casting operators....113
Literals....114
Promotion....116
C# has **sizeof**....116
C#'s preprocessor....116
Precedence revisited....118
A compendium of operators....118
Execution control....130
true and false....130
if-else....130
return....131
Iteration....132
do-while....133
for....133
foreach....135
The comma operator....136
break and **continue**....136
The infamous **goto**....138
switch....144
Summary....146
Exercises....146
5: Initialization and Cleanup....149
Guaranteed initialization with the constructor....149
Method overloading....152
Distinguishing overloaded methods....154
Overloading with primitives....155
Overloading on return values....159
Default constructors....159
The **this** keyword....160
Calling constructors from constructors....164
The meaning of **static**....165
Cleanup: finalization and garbage collection....166
What are destructors for?....167
Instead of a destructor, implement **IDisposable.Dispose( )**....168

Destructors, **IDisposable**, and the **using** keyword....173
How a garbage collector works...174
Member initialization....176
Specifying initialization....178
Constructor initialization....179
Array initialization....185
The **params** method modifier..189
Multidimensional arrays....190
What a difference a rectangle makes194
Summary....195
Exercises....196

6: Hiding the Implementation

Organizing with namespaces200
Creating unique names....202
Using **#define** to change behavior204
C#'s access specifiers....206
public: interface access....206
internal....207
private: you can't touch that!...208
protected....210
Interface and implementation211
Class access....213
Summary....215
Exercises....217

7: Reusing Classes

Composition syntax....220
Inheritance syntax....223
Initializing the base class....225
Combining composition and inheritance....229
Guaranteeing proper cleanup....230
Choosing composition vs. inheritance....234
protected....235
Incremental development...236
Upcasting....237
Why "upcasting"?....238
Explicit overloading only....239
The **const** and **readonly** keywords....251
Sealed classes....253
Emphasize virtual functions....254
Initialization and class loading255
Initialization with inheritance....255
Summary....257
Exercises....258

8: Interfaces and Implementation

Upcasting revisited....262
Forgetting the object type....263
The twist....263
Method-call binding....265
Producing the right behavior....266
Extensibility....270
Static methods cannot be virtual....274
Overriding vs. overloading..276
Operator overloading....278
Multiargument operator overloading....282
Explicit and implicit type conversions....283
Operator overloading design guidelines....285
Abstract classes and methods285
Constructors and polymorphism....289
Order of constructor calls....290
Behavior of polymorphic methods inside constructors....292
Designing with inheritance.294
Pure inheritance vs. extension...295
Downcasting and run-time type identification....298
Interfaces....302
"Multiple inheritance" in C#....305

Extending an interface with inheritance 309
Summary 310
Exercises 311

9: Coupling and Cohesion 315
Software as architecture vs. software architecture 315
What is software architecture? 317
Simulation architectures: always taught, rarely used ... 318
Client/server and *n*-tier architectures 318
Layered architectures 322
Problem-solving architectures 323
Dispatching architectures 323
"Not really object-oriented" 324
Coupling 324
Cohesion 329
Design is as design does 339
First, do no harm 339
Write boring code 340
Make names meaningful 340
Limit complexity 341
Make stuff as private as possible 343
Coupling, cohesion, and design trends 344
Summary 345
Exercises 346

10: Collecting Your Objects 349
Arrays 349
Arrays are first-class objects 351
The **Array** class 355
Array's static methods 355
Array element comparisons 358
What? No bubbles? 360
Unsafe arrays 362
Get things right... 366
... then get them fast 368
Array summary 375
Introduction to data structures 380
Queues and stacks 381
ArrayList 384
BitArray 386
Dictionaries 387
Hashtable 388
ListDictionary 391
SortedList 391
String specialists 392
One key, multiple values 392
Customizing hashcode providers 394
String specialists: **StringCollection** and **StringDictionary** 396
Container disadvantage: unknown type 397
Using **CollectionBase** to make type-conscious collections 399
IEnumerators 401
Custom indexers 403
Custom enumerators & data structures 406
Sorting and searching **List**s 413
From collections to arrays ... 414
Persistent data with ADO.NET 421
Getting a handle on data with **DataSet** 422
Connecting to a database 425
Fast reading with **IDataReader** 428
CRUD with ADO.NET 430
Update and delete 430
The object-relational impedance mismatch 434
Summary 435
Exercises 436

11: Error Handling with Exceptions
Basic exceptions 441
Exception arguments 442
Catching an exception 443

The **try** block..... 443
Exception handlers 443
Supertype matching..... 444
Exceptions have a helplink 444
Creating your own exceptions445
C#'s lack of checked exceptions451
Catching any exception..... 452
Rethrowing an exception 453
Elevating the abstraction level... 453
Standard C# exceptions..... 455
Performing cleanup with **finally** 456
What's **finally** for? 457
Finally and **using** 461
Pitfall: the lost exception 462
Constructors 464
Exception matching.....468
Exception guidelines..... 469
Summary.....470
Exercises 470

12: I/O in C#

File, **Directory**, and **Path** 473
A directory lister 473
Checking for and creating directories 474
Isolated stores 476
Input and output 478
Types of **Stream** 479
Text and binary 479
Working with different sources .480
Fun with **CryptoStream**s..... 482
BinaryReader and **BinaryWriter**..... 486
StreamReader and **StreamWriter**.....491
Random access with Seek....494
Standard I/O.....495
Reading from standard input 496
Redirecting standard I/O..... 496
Debugging and Tracing..... 497
Regular expressions 499
Checking capitalization style.....504
Summary.....508
Exercises 508

13: Reflection and Attributes

The need for RTTI 511
The **Type** object 514
Checking before a cast..... 517
RTTI syntax 523
Reflection: run-time class information..... 525
Adding meta-information with attributes 527
Attributes are just classes..... 527
Specifying an attribute's targets.528
Attribute arguments 529
The Global Assembly Cache 532
Designing with attributes..... 537
Beyond objects with aspects.....543
Summary..... 543
Exercises 473

14: Programming Windows Forms

Delegates.....548
Designing With Delegates ...550
Multicast delegates..... 552
Events 555
Recursive traps 558
The genesis of Windows Forms561
Creating a **Form** 562
GUI architectures 563
Using the Visual Designer ... 563
Form-Event-Control..... 570
Presentation-Abstraction-Control 573
Model-View-Controller 577
Layout 582
Non-code resources 585
Creating satellite assemblies 590

Constant resources 591
What about the XP look?..... 593
Fancy buttons 596
Tooltips 599
Displaying and editing text..600
Linking text.....604
Checkboxes and **RadioButtons** 606
List, **Combo**, and **CheckedListBoxes**.....609
Multiplane displays with the **Splitter** control 615
TreeView and **ListView**... 616
ListView..... 618
Icon views..... 618
Details view..... 618
Using the clipboard and drag and drop.....622
Clipboard..... 622
Drag and drop 624
Data-bound controls..... 634
Editing data from bound controls 639
Menus 646
Standard dialogs.....650
Usage-centered design 653
Summary..... 654
Exercises 655

15: GDI+ Overview
Your canvas: the **Graphics** Class 659
Understanding repaints..... 661
Control: paint thyself.....662
Scaling and transforms..... 665
Filling regions 672
Non-rectangular windows... 677
Matrix transforms.....678
Hit detection 686
Fonts and text 688
Printing 690
Bitmaps 692
Rich clients with interop 698
COM Interop and the **WebBrowser** control.....698
COM Interop challenges..... 701
Non-COM Interop 702
Summary..... 707
Exercises 708

16: Multithreaded Programming
Responsive user interfaces ... 711
.NET's threading model.....714
Running a thread..... 716
Waiting for a thread to complete 717
Multiple threads in action 719
Threading for a responsive interface721
Interrupting a sleeping **Thread** 724
Thread.Join( ) waits for another thread to end..... 727
Sharing limited resources.... 729
Improperly accessing resources .730
Using **Monitor** to prevent collisions 735
lock blocks – a shortcut for using **Monitor**..... 741
Monitoring static value types 748
The Monitor is not "stack-proof" 751
Cross-process synchronization with **Mutex** 752
Deadlocks..... 753
Not advised: **Suspend( )** and **Resume( )**.....760
Threads and collections..... 765
Summary..... 769
Exercises 770

17: XML
XML structure 774
XML as a stream..... 775
XML as a tree.....777

Writing XML 778
XML serialization 783
Deserializing XML 789
Can't serialize cycles 789
Schemas 796
ADO and XML 798
XPath navigation 801
An XPath explorer 807
Transforming a document ... 815
Summary 821
Exercises 822

18: Web Programming
Identifying a machine 824
Sockets 826
Whois for ZoneAlarm 826
Receiving incoming connections 833
Serving multiple clients 837
Communicating with Microsoft Messenger 841
Creating and receiving HTTP requests 852
Asynchronous Web requests 858
From Web programming to Web Services 864
Insanely simple Web services 865
Maintaining state 868
Web services vs. Web APIs 868
Consuming Web services 872
Modifying XML returns 874
Summary 876
Exercises 877

A: C# for Visual Basic Programmers

B: C# for Java Programmers

C: Test-First Programming with NUnit

D: Programming the Pocket PC

E: C# programming guidelines
Design 903
Implementation 910

F: Resources
.NET Software 915
Non-.NET Books 915

Concordance

Class, Method, Property Cross-Reference

Index

Preface: Computer Language

C# is a language, and like all languages, therefore a way of thinking. Languages channel us down particular avenues of thought, make certain ideas as obvious as a grand concourse and others as confusing and mysterious as a back alley. Different computer languages facilitate different things; there are computer languages that facilitate graphics programming and others that are best for text manipulation, many that excel in data relationships, and several whose raison d'être is pure performance. C# is a language for professional programming. The ideas that it facilitates, the capabilities that it makes easy, are those that lead to the rapid development of robust, scalable programs that deliver client value and are easily modifiable.

You can't look at C# as just a list of keywords that must be memorized to get a clean compile or as a conveyor belt for calling library functions. You must look at it as an interlocking set of features that support the efficient creation of object-oriented, high-quality programs. And to understand C# in this way, you must understand both its strengths and its weaknesses, and how they relate to the best practices that are known for developing software and the challenges that remain. This book discusses programming as a profession, not as an academic discipline, and the pragmatic use of C# and the .NET Framework SDK. Thus, the chapters present their features based on what we the authors believe to be the core issues of the subject and the way in which C# addresses those issues.

A chapter in *Thinking in C#* should take you to the point where you can take charge of your own further education by whatever means you find most constructive. For some topics, you may find the background provided by the book sufficient and concern yourself only with fleshing out the details of the .NET

library classes and methods in the area. Hopefully, some topics will excite your interest and you will seek out a deeper understanding of the underlying principles.

Every chapter in this book is worthy of book-length discussion and *Thinking in C#* necessarily glosses over many issues. Rather than hide these decisions in academic rhetoric, this book tries to make explicit the subjective opinions of the authors, Larry O'Brien and Bruce Eckel. Additionally, neither of us is in the employ of Microsoft[1] and both of us are fairly jaded when it comes to languages, frameworks, and implementations. We do not hesitate to criticize design decisions with which we disagree nor do we pause when it comes to crediting Java as an important influence as both a marketing and technical influence on C# and .NET. Both of us have been programming since the 1970s and writing and teaching on these subjects since the '80s, so our opinions may be judged incorrect, but we come to those opinions by experience.

Prerequisites

This book assumes that you have some programming familiarity: you understand that a program is a collection of statements, the idea of a subroutine/function/macro, control statements such as "if" and looping constructs such as "while," etc. However, you might have learned this in many places, such as programming with Microsoft Office's Visual Basic for Applications or working with a tool like Perl. As long as you've programmed to the point where you feel comfortable with the basic ideas of programming, you'll be able to work through this book.

The book will be easiest for Visual Basic, Delphi, or Java programmers. Visual Basic programmers will be familiar with many library names and several of the programming models, Delphi programmers will recognize in C# the influence of Anders Hejlsberg, and Java programmers will find the hardest thing about moving to C# is getting used to a different naming convention. If you don't have a background in those languages, don't count yourself out, although naturally it means that you will be required to expend a little more effort.

This book does *not* assume that you're familiar with object-oriented programming (OOP) and the first half of the book can be seen as an extended tutorial on object-oriented programming at least as much as a tutorial on C# per se. No formal background in computer science is expected.

---

[1] Larry has been paid to write technical white papers for Microsoft.

Although references will often be made to language features in other languages such as C++ and Java, these are not intended to be insider comments, but instead to help all programmers put C# in perspective with those languages. All good programmers are polyglots and the greatest value proposition of the .NET Framework is that it supports multiple languages.

Learning C#

Picasso is reputed to have said "Computers are worthless; they can only give you answers." The same could be said of books. No book can teach you C# programming, because programming is a creative process. The only way to learn any language is to use it in a variety of situations, to gradually internalize it as you solve increasingly difficult problems with it. To learn C#, you must start programming in C#.

This is not to say that the best way to learn C# is on the job. For one thing, companies don't typically allow programmers to work in a language in which they have no experience. More significantly, your *job* is to deliver value to your customers, not to learn the nuances of C# and the .NET Framework.

For many people, seminars are the best environment for rapid learning. There are many reasons for this: the interactions with the teachers and fellow students, the explicit dedication of several days to achieving specific educational goals, or just being out of the office and away from email and meetings. One of the authors (Bruce) has been teaching object-oriented programming in multiday seminars since 1989. The structure of this book is highly influenced by those experiences.

As the chair of the C++ and Java tracks at the Software Development conference, Bruce discovered that speakers tended to give the typical audience too many topics at once. Sometimes this was because they were striving to present an example that was "realistic" and therefore unfocused. Other times it stemmed from a fear of underserving the more experienced in the audience. Over the years, Bruce developed many presentations, iteratively developing a scope and sequence for teaching object-oriented programming in a language-specific manner. This curriculum has been the core of many products: books, CD-ROMs, and seminars for a variety of languages including C++, Java, and, now, C#.

There are three notable characteristics of this curriculum:

- It has a broad scope, from fundamental topics such as "how does one compile a program?" to professional challenges such as thread-safe design

- It is outcome-oriented: the goal is to give the learner the basic skills for professional object-oriented development in the language
- It is dependent on the learner's active engagement with the samples and exercises

Although this book is influenced by the seminars and books that preceded it, it is not just seminar notes. More than anything, the book is designed to serve the solitary reader who is struggling with a new programming language.

Goals

Like its immediate predecessor *Thinking in Java*, this book is structured around the process of teaching the language. In particular, the structure is based on the way the language can be taught in seminars. Chapters in the book correspond to what experience has taught is a good lesson during a seminar. The goal is to get bite-sized pieces that can be taught in a reasonable amount of time, followed by exercises that are feasible to accomplish in a classroom situation.

The goals of this book are to:

1. Present the material one simple step at a time so that you can easily digest each concept before moving on.

2. Use examples that are as simple and short as possible. This generally prevents "real world" problems, but it's better to understand every detail of an example rather than being impressed by the scope of the problem it solves.

3. Carefully sequence the presentation of features so that you aren't seeing something that you haven't been exposed to. Of course, this isn't always possible; in those situations, a brief introductory description is given.

4. Give a pragmatic understanding of the topic, rather than a comprehensive reference. There is an information importance hierarchy, and there are some facts that 95 percent of programmers will never need to know and that just confuse people and add to their perception of the complexity of the language. To take an example from C#, if you memorize the operator precedence table on page 118, you can write clever code. But if you need to think about it, it will also confuse the reader/maintainer of that code. So forget about precedence, and use parentheses when things aren't clear.

5. Keep each section focused enough so that the range of topics covered is digestible. Not only does this keep the audience's minds more active and

involved during a hands-on seminar, but it gives the reader a chance to tackle the book within the busy time constraints that we all struggle with.

6. Provide you with a solid foundation so that you can understand the issues well enough to move on to more difficult coursework and books.

Online documentation

The .NET Framework SDK (a free download from Microsoft) comes with documentation in Windows Help format. So the details of every namespace, class, method, and property referenced in this book can be rapidly accessed simply by working in the Index tab. These details are usually not discussed in the examples in this book, unless the description is important to understanding the particular example.

Exercises

Exercises are a critical step to internalizing a topic; one often believes that one "gets" a subject only to be humbled doing a "simple exercise." Most exercises are designed to be easy enough that they can be finished in a reasonable amount of time in a classroom situation while the instructor observes, making sure that all the students are absorbing the material. Some exercises are more advanced to prevent boredom for experienced students.

The first half of the book includes a series of exercises that are designed to be tackled by iterative and incremental effort—the way that real software is developed. The second half of the book includes open-ended challenges that cannot be reduced to code in a matter of hours and code, but rather are intended to challenge the learner's synthesis and evaluation skills[2].

Source code

All the source code for this book is available as copyrighted freeware, distributed as a single package, by visiting the Web site *www.ThinkingIn.Net*. To make sure that you get the most current version, this is the official site for distribution of the code and the electronic version of the book. You can find mirrored versions of the electronic book and the code on other sites (some of these sites are found at *www.ThinkingIn.Net*), but you should check the official site to ensure that the

[2] Professional educators should contact the authors for a curriculum including pre- and post test evaluation criteria and sample solutions.

mirrored version is actually the most recent edition. You may distribute the code in classroom and other educational situations.

The primary goal of the copyright is to ensure that the source of the code is properly cited, and to prevent you from republishing the code in print media without permission. (As long as the source is cited, using examples from the book in most media is generally not a problem.)

In each source code file you will find a reference to the following copyright notice:

```
//:! :Copyright.txt
Copyright ©2002 Larry O'Brien
Source code file from the 1st edition of the book
"Thinking in C#." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in C#" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.ThinkingIn.Net
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Larry O'Brien makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Larry O'Brien, Bruce Eckel, and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
```

```
Larry O'Brien, Bruce Eckel or the publisher be liable for
any lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Larry O'Brien, Bruce
Eckel and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.ThinkingIn.Net. (Please use the same
form for non-code errors found in the book.)
///:~
```

You may use the code in your projects and in the classroom (including your presentation materials) as long as the copyright notice that appears in each source file is retained.

Coding standards

In the text of this book, identifiers (function, variable, and class names) are set in **bold**. Most keywords are also set in bold, except for those keywords that are used so much that the bolding can become tedious, such as "class." Important technical terms (such as *coupling*) are set in italics the first time they are used.

The coding style used in this book is highly constrained by the medium. Pixels are cheap; paper isn't. The subject of source-code formatting is good for hours of hot debate, so suffice it to say that the formatting used in this book is specific to the goals of the book. Since C# is a free-form programming language, you and your teammates can use whatever style you decide is best for you.

The programs in this book are files that are included by the word processor in the text, directly from compiled files. Thus, the code files printed in the book should all work without compiler errors. The errors that *should* cause compile-time error messages are commented out with the comment **//!** so they can be easily discovered and tested using automatic means. Errors discovered and reported to the author will appear first in the distributed source code and later in updates of the book (which will also appear on the Web site *www.ThinkingIn.Net*).

C# and .NET versions

All of the code in this book compiles and runs with Microsoft's .NET Framework 1.1.4322and Microsoft's Visual C# .NET Compiler 7.10.2215.1, which were released in the Fall of 2002.

Seminars and mentoring

Bruce Eckel's company MindView provides a wide variety of learning experiences, ranging from multiday in-house and public seminars to get-togethers whose goal is to facilitate the "hallway conversations" that are so often the place in which great leaps in understanding and innovation take place. Larry O'Brien teaches seminars, but is more often engaged as a direct mentor and active participant in programming projects. You can sign up for an occasional announcement newsletter on upcoming C# and .NET learning experiences at *www.ThinkingIn.Net*.

Errors

No matter how many tricks a writer uses to detect errors, some always creep in and these often leap off the page for a fresh reader.

If you discover anything you believe to be an error, please send an email to *corrections@ThinkingIn.Net* with a description of the error along with your suggested correction. If necessary, include the original source file and note any suggested modifications. Your help is appreciated.

Note on the cover design

The cover of *Thinking in C#* portrays a kelp bass (*Paralabrax clathratus*), a vermilion rockfish (*Sebastes miniatus*), and a trio of kelp greenling (*Hexagrammos decagrammus*), three species that might be encountered while SCUBA diving in California's kelp forests. Like programming, SCUBA diving is an activity dependent on technology. Just as the real joy of SCUBA diving does not reside in the technology but in the realm the technology opens, so too is it with computer programming. Yes, you must become familiar with a technology and some principles that may seem arcane at first, but eventually these things become second nature and a world that cannot be appreciated by non-practitioners opens up to you.

People who have splashed around with a mask, snorkeled off a sandy beach, and watched Shark Week on The Discovery Channel have little or no concept of the great privilege it is to enter a realm only recently available to humanity. People

who just use computers to send email, play videogames, and surf the Internet are missing their opportunity to actively participate in the opening of a great intellectual frontier.

Acknowledgments

(by Larry O'Brien)

First, I have to thank Bruce Eckel for entrusting me to work with the *Thinking In…* structure. Without this proven framework, it would have been folly to attempt a work of this scope on a brand-new programming language.

I'm going to exercise my first-time book author's perquisite to reach back in time to thank J.D. Hildebrand, Regina Ridley, and Don Pazour for hiring a blatantly unqualified hacker with a penchant for Ultimate Frisbee and giving me the greatest programming job in the world – Product Review Editor of *Computer Language* magazine. For half a decade I had the ridiculous privilege of being able to ask many of the brightest and most interesting people in the software development industry to explain things in terms that I could understand. It would be folly to try to begin to list the writers, readers, editors, speakers, and students to whom I am indebted, but I have to thank P.J. Plauger and Michael Abrash for demonstrating a level of readability and technical quality that is inspiring to this day and Stan Kelly-Bootle for his trail-blazing work in developing a programmer's lifestyle worthy of emulation (e.g., at your 70th birthday party there should be an equal mix of language designers, patrons of the symphony, and soccer hooligans).

Alan Zeichick urged me to write a book on C#, a display of considerable faith considering the number of times I have missed deadlines on 1,000-word articles for him. Claudette Moore and Debbie McKenna of Moore Literary Agency were tremendously helpful in representing me and Paul Petralia of Prentice Hall always agreed that the quality of the book took precedence over schedule pressure. Mark Welsh's commitment to the book even after his internship ended is something for future employers to note.

A draft of the book was made available on the Internet and was downloaded on the order of 100,000 times. The integrated Backtalk system that allowed paragraph-by-paragraph feedback from readers was developed by Bruce and allowed far more people than can be listed to contribute to the book. The contributions of Bob Desinger, Michel Lamsoul, and Edward Tanguay were especially beneficial. Reg Charney and members of the Silicon Valley C/C++ User's Group contributed greatly to the discussion of deterministic finalization, a

subject also frequently visited on DevelopMentor's excellent .NET and C# discussion lists and the GotDotNet.com forums.

Eric Gunnerson of Microsoft's C# team gave enormously valuable feedback, particularly in areas speaking to the intent of the language designers; if the book is unfair to C# or Microsoft or misstates reasoning, the fault lies solely in the authors' pig-headedness. It is an open secret that Microsoft's public relations firm of Waggener Edstrom is one of the keys to Microsoft's success; Sue Schmitz's responsiveness during the writing of this book was stellar even by WaggEd's high standards.

C.J. Villa, Ben Rafter, and Ken Bannister pointedly ignored the times when I let book issues interfere with my work. I too often bore my non-programming friends with tales of technical drama, but Chris Brignetti and Sarah Winarske have been especially stoic over the years. Dave Sieks has kept me laughing since fifth grade, where he demonstrated prior art that should invalidate U.S. patent number 6,368,227. Finally, I have to thank the crew of the diveboat FeBrina for reminding me that technology is just a way to get to the good stuff.

1: Those Who Can, Code

Computer programming is tremendous fun. Like music, it is a skill that derives from an unknown blend of innate talent and constant practice. Like drawing, it can be shaped to a variety of ends – commercial, artistic, and pure entertainment. Programmers have a well-deserved reputation for working long hours but are rarely credited with being driven by creative fevers. Programmers talk about software development on weekends, vacations, and over meals not because they lack imagination, but because their imagination reveals worlds that others cannot see.

Programming is also a skill that forms the basis of one of the few professions that is consistently in high demand, pays fairly well, allows for flexibility in location and working hours, and which prides itself on rewarding merit, not circumstances of birth. Not every talented programmer is employed, women are under-represented in management, and development teams are not color-blind utopias. But on the whole, software development is a very good career choice.

Coding, the line-by-line development of precise instructions for the machine to execute, is and will always be the core activity of software development. We can say this with certainty because no matter what happens in terms of specification languages, probabilistic inference, and computer intelligence, it will always be painstaking work to remove the ambiguity from a statement of *client value*. Ambiguity itself is enormously valuable to humans ("That's beautiful!" "You can't miss the turn-off," "With liberty and justice for all") and software development, like crafting legal documents, is a task where the details are necessarily given a prominence that is quite the opposite of how people prefer to communicate.

This is not to say that coding will always consist of writing highly structured lines of text. The *Uniform Modeling Language* (UML), which specifies the syntax and semantics of a number of diagrams appropriate to different software

development tasks, is expressive enough that one *could* code in it. But doing so is hugely inefficient when compared to writing lines of text. On the other hand, a single UML diagram can clarify in a moment structural and temporal relationships that would take minutes or hours to comprehend with a text editor or a debugger. It is a certainty that as software systems continue to grow in complexity, *no* single representation will prove universally efficient. But the task of removing ambiguity, task-by-task, step-by-step, will always be a time-consuming, error-prone process whose efficiency is reliant on the talents of one or more programmers.

There is more to professional software development than writing code. Computer programs are among the most complex structures ever constructed by humanity and the challenges of communicating desires and constraints, prioritizing effort, managing risk, and above all, maintaining a working environment that attracts the best people and brings forth their greatest efforts… well, software project management takes a rare combination of skills, skills that are perhaps rarer than the skills associated with coding. All good programmers discover this eventually (almost always sooner rather than later) and the best programmers inevitably develop strong opinions about the software development process and how it is best done. They become team leads and architects, engineering managers and CTOs, and as these elite programmers challenge themselves with these tasks, they sometimes forget about or dismiss as trivial the challenges that arise between the brackets of a function.

This book stops at the screen edge. That isn't a judgment that issues of modeling and process and teamwork aren't as important as the task of coding; we the authors know that these things are at least as important to the development of successful products as coding. But so is marketing. The tasks that *are* discussed in this book, the concerns of professional coding, are not often written about in a language-specific manner.

One reason that the *concerns* of coding (as opposed to the mere *details* of coding) are rarely discussed in a language-specific way is that it is almost impossible to assume anything about the background of a person choosing to program in C#. C# has a lot of appeal to younger programmers, as it is the simplest (and potentially free[1]) route to the full capabilities of the broadest range of computers, while older programmers will find in C# the ideal opportunity to mitigate the risk of obsolescence while maintaining (and, after the initial learning period,

[1] All programs in this book can be written, compiled, and run with tools that are available for no charge from Microsoft and others. See http://www.ThinkingIn.Net/tools.html

increasing) productivity. Java programmers sick of the complexity of J2EE or frustrated by the lack of OS integration will be thrilled by the productivity of the .NET Framework while Visual Basic programmers have in C# the ideal opportunity to move into the mainstream of languages derived from the C syntax. C# could even be a good stepping stone *towards* Java for those programmers who wish to maintain the widest possible base of expertise.

Since it's impossible for us to assume anything about your background, we instead assume several things about your skills and motivation. This book constantly shifts the level of discourse from details to theory and back down to details, a technique that is patently inappropriate for some learners. Rapid shifts of abstraction levels are part and parcel of software development, however. Most programmers can relate to the experience of a high-flying business meeting with discussion of "synergy" and "new paradigms" and "money dripping from the ceiling" being suddenly interrupted by a programmer who skeptically says "Now wait a second…" and says something incomprehensible to non-programmers. Then some other programmer says something equally incomprehensible in response. This "speaking in binary" goes back and forth for a minute or so and the skeptical programmer suddenly rocks back and declares to the businesspeople: "Oh, you don't even *get* how huge this is!"

This is not a book about shortcuts and getting by, it is a book about tackling hard problems in a professional manner. In keeping with that, *Thinking in C#* accelerates the pace of discussion throughout the book. An issue that earlier in the book was the subject of pages and pages of discussion may be referred to off-handedly or even go unremarked in later chapters. By the time you're using C# to develop Web Services professionally, you *must* be able to discuss object-oriented design at the level in which it is presented in that chapter.

To understand why C# and .NET succeed at a *programming* level, though, it's important to understand how they succeed at the *business* level, which means discussing the economics of software development.

Software economics

Ever since Alan Turing introduced the concept of a universal computer and then John von Neumann the idea of a stored program, software developers have struggled to balance the symbol-manipulating power of increasing levels of abstraction with the physical constraints of speed, storage, and transmission-channel capacity of the machine at hand. There are people still alive who can talk about reading the output of the most sophisticated computer in existence by holding a cardboard ruler up to an oscilloscope and judging the signal as either a

one or a zero. As recently as the early 1980s, the coolest computers in widespread circulation (the DEC PDP series) could be programmed by flipping switches on a panel, directly manipulating chip-level signals. And until the 1990s, it was inconceivable for a PC programmer to be unfamiliar with a wide range of interrupt requests and the memory addresses of various facilities.

Between the mid-1960s, when the IBM 360 was released and Gordon Moore formulated his famous law that transistor density in a given area would continue to double every 18 months, the cost of a single processing instruction has decreased by approximately 99.99%. This has totally inverted the economics of programming. Where once it made sense for the programmer to work with a mental model of the computer's internal representation and to sacrifice development time for execution efficiency, now it makes sense for the computer to be given a model corresponding to the programmer's internal representation of the task, even if that representation is not ideally efficient.

Today, the quality of a programming language can be judged by how easily one can express the widest variety of problems and solutions. By that standard, *object-oriented, imperative programming languages* absolutely dominate the world of software development. An *imperative* language is one in which a series of commands is given to the computer: do this, then do that, then do this other thing. The imperative may seem like the "natural" way to program computers in that it corresponds to our mental model of how computers work, but as discussed above, this is no longer a very good reason to embrace a programming language. Think about how easy some problems are to solve with a spreadsheet, which can be viewed as a form of non-imperative programming. However, imperative programming is deeply ingrained in the mainstream zeitgeist and is unlikely to be dethroned anytime soon.

C# and the Internet

Since the mid-'90s, the world of programming has been transformed. Prior to the explosive growth of the Internet, most programs were written for either internal consumption within an organization or were "shrink-wrapped" applications that provided some type of generic service for a faceless customer. With the rise of the Web, though, a vast amount of programming effort has shifted to directly delivering value to the customer. Value on the Web takes many forms: lower prices (although the days of below-wholesale costs and free giveaways seem to have passed), convenience, access to greater inventory, customization, collaboration, and timeliness are just some of the true values that can be derived from Web-based services.

Static sites

Even the simplest business site requires some programming to handle Web form input. While these can often be handled by a scripting language such as Perl, Perl doesn't integrate into a Windows-based server as well as it does into UNIX (many Perl scripts downloadable from the Web assume the availability of various UNIX facilities such as **sendmail**). The .Net Frameworks **IHttpHandler** class allows a straightforward and clean method of creating simple form-handlers while also providing a path towards much more sophisticated systems with complex designs.

Dynamic sites

ASP.NET is a complete system for creating pages whose contents dynamically change over time and is ideal for eCommerce, customer-relations management, and other types of highly dynamic Web sites. The idea of "active server pages" which combine programming commands and HTML-based display commands was originally perceived as a bridge between Web designers trained in graphic arts and the more disciplined world of programming. Instead, server-page programming evolved into a for-programmers technology that is now widely used as the model for complete Web solutions.

Server-page programming, like Visual Basic's form-based programming model, facilitates the intertwining of display and business-logic concerns. This book promotes the view that such intertwining is ill-advised for non-trivial solutions. This doesn't mean that ASP.NET and Visual Basic are poor languages; quite the opposite, it means that their programming models are so flexible that doing great work in them actually requires *more* understanding and discipline than is required with C#.

Peer-to-peer

One of the last dot-com technology-hype-of-the-month phrases was *peer-to-peer* (also known as P2P, which had the hype advantage of being the same acronym as one of the last business-hype-of-the-month phrases *path-to-profitability*). Ironically, P2P is the type of architecture that one would expect from the phrase World Wide "Web." In a P2P architecture, services are created in two steps: peer resources are discovered by some form of centralized server (even if the server is not under the legal control of the coordinating organization) and then the peers are connected for resource sharing without further mediation.

C# and .NET have strong facilities for the creation of P2P systems, as such systems require the creation of rich clients, sophisticated servers, and facilities for creating robust resource-sharing systems. P2P is tainted by the prevalence of

file-sharing systems, but programs such as SETI@Home and Folding@Home demonstrate the potential for grid computing, which can bring staggering amounts of computation to bear on challenging problems.

Web services

The value that has been created from a foundation of HTML is astonishing. The value that will be created from the far more flexible and expressive Extensible Markup Language (XML) will out-strip everything that has gone before (maybe not in terms of stock prices and company valuations, but in actual productivity and efficiency, there is no question). Web Services deliver value by standard Web protocols and XML-based data representation that does not concern itself with how it is displayed (Web Services are *headless*).

Web Services are the raison d'être of Microsoft's entire .NET strategy, which is considerably broader than "just" the biggest update in programming APIs in a decade. .NET is wrongly interpreted by many business writers as an attempt by Microsoft to introduce itself as a central mediator in over-the-Web transactions. The reality is simpler; Microsoft wants to own the operating systems on all Web-connected devices, even as the type and number of such devices skyrocket. The more that computers shift from performing primarily computational tasks towards communication and control tasks, the more that Web Services have to offer, and Microsoft has *always* understood that operating system dominance is controlled by software development.

The .NET strategy is an across-the-board shift towards a post-desktop reality for Microsoft and software development. The .NET Framework, which combines an abstraction of the underlying hardware with comprehensive Application Programming Interfaces (APIs), proposes to developers that "write once, run anywhere" is an anachronistic view that promotes the concept of *a* component running on *a* computer. The .NET strategy recognizes that rich clients ("rich clients" meaning non-server applications that are responsible for computing more than simply their display and input functions) operating on a variety of devices, high-performance servers, and new applications running on the not-to-be-abandoned desktop "legacy" machines are not *separate* markets at all, but rather are components that all software applications will necessarily have to address. Even if programmers begin with a browser-based client for their Web Service (or, for that matter, even if programmers develop a Windows-based rich client for a Java-based Web Service), part of the .NET strategy is to make it unfailingly easy to extend the value on another device: a rich client on the PocketPC or 3G phone or a high-performance database in a backend rack. Web protocols will connect all these devices, but the *value* is in the information, which

will flow via Web Services. If .NET is the most expedient way to develop Web Services, Microsoft will inevitably gain marketshare across the whole spectrum of devices.

Security

The quality of Microsoft's programming is often judged unfairly. No operating system but Windows is judged by what esoteric hardware configurations it *doesn't* run on and Microsoft Office may take up a disconcerting amount of disk space to install, but it's reliable and capable enough to monopolize the business world. But where Microsoft has legitimately goofed up is in security. It's bad enough that Microsoft generally makes security an all-or-nothing decision ("Enable macros, yes or no?" "Install this control (permanently), yes or no?") but the fact that they give you no data for that all-or-nothing decision ("Be sure that you trust the sender!") is unforgiveable. When you consider the number of files that have been transferred on and off the average computer and the lack of sophistication of many users, the only thing that's surprising is how rare truly devastating attacks have been.

The .NET Framework SDK includes a new security model based on fine-grained permissions for such things as accessing the file system or the network and digital signatures based on public-key cryptography and certificate chains. While Microsoft's stated goal of "trustworthy computing" goes beyond security and will require significant modifications in both their operating systems and, perhaps even more critically, directly in Microsoft Office and Outlook, the .NET Framework SDK provides sophisticated components which one can imagine giving rise to a much more secure computing environment.

Analysis, design, and Extreme Programming

Only about a fourth of professional software is developed on time[2]. This is due to many reasons, including a quirk in programmers' psychology that can only be described as "irrational over-confidence." The most significant contributors to time-and-cost overruns, though, are failures in the discovery and description of

[2] This is a rough figure, but widely supported. Capers Jones, Barry Boehm, and Steve McConnell are trustworthy names in the area of software project economics, which otherwise is dominated by conjecture and anecdote.

users' needs and the negotiation of software features to answer those needs. These processes are called, respectively, *analysis* and *design*[3].

Cost-and-time overruns are driven by a few underlying truths:

- Software project estimation is done haphazardly
- Communication is fraught with misunderstanding
- Needs change over time
- It is difficult to visualize and understand the interactions of complex software systems
- People tend to advocate the things in which they've already invested
- Computers do what you say, not what you mean

On a practical basis, overruns occur because all sorts of assumptions about scope, personnel, and system behavior get turned into some kind of rough plan that is then converted into a formal commitment by wishful thinking, financial imperatives, and a Machiavellian calculation to exploit the common wisdom that "no one can predict software costs" to avoid responsibility down the line.

In a small program that is only a few thousand lines of code, these issues don't play a major role and the majority of the total effort is expended on *software construction* (detailed design, coding, unit testing, and debugging). In larger programs (and many corporations have codebases of several hundred thousand or even millions of lines of code), the costs of analysis, design, and integrating the new code into the old (expensive because of unexpected side-effects) have traditionally outstripped the costs of construction.

Recently, the programming world has been shaken by a set of practices that basically turn big projects into a series of small projects. These *Extreme Programming* (XP[4]) practices emphasize very close collaboration and dramatically reduced product lifecycles (both in the scope of features released and the time between releases). XP's most famous and controversial practice is "pair programming," in which two programmers literally share a monitor and keyboard, reversing the stereotype of the lone programmer entranced with his or

---

[3] It is often helpful to distinguish between *high-level design* that is likely to have meaning to the user and *low-level design* that consists of the myriad technical decisions that are made by the programmer but which would be incomprehensible to the user.

[4] Not to be confused in any way with Windows XP.

her singular work[5]. Where traditional software releases have been driven by 12-, 18-, and 24-month release cycles, XP advocates propose 2- and 4-week release cycles.

C#, .NET, and Visual Studio .NET do not have any special support for either Extreme Programming or more formal methodologies. Both authors' experiences make us strong advocates of XP or XP-like methods and as this book is unabashedly subjective and pragmatic, we advocate XP practices such as unit testing throughout. Appendix C, "Test-First Programming with NUnit," describes a popular unit-testing framework for .NET.

Strategies for transition

Here are some guidelines to consider when making the transition to .NET and C#:

1. Training

The first step is some form of education. Remember the company's investment in code, and try not to throw everything into disarray for six to nine months while everyone puzzles over how interfaces work. Pick a small group for indoctrination, preferably one composed of people who are curious, work well together, and can function as their own support network while they're learning C# and .NET.

An alternative approach that is sometimes suggested is the education of all company levels at once, including overview courses for strategic managers as well as design and programming courses for project builders. This is especially good for smaller companies making fundamental shifts in the way they do things, or at the division level of larger companies. Because the cost is higher, however, some may choose to start with project-level training, do a pilot project (possibly with an outside mentor), and let the project team become the teachers for the rest of the company.

2. Low-risk project

Try a low-risk, low-complexity project first and allow for mistakes. The failure rate of first-time object-oriented programs is approximately 50%[6]. Once you've gained some experience, you can either seed other projects from members of this

---

[5] Unfortunately for XP, a lot of programmers embrace the stereotype and are not interested or willing to share their keyboards.

[6] *Software Assessments, Benchmarks, and Best Practices*, Capers Jones, 2000, Addison-Wesley (ISBN: 0-201-48542-7).

first team or use the team members as a .NET technical support staff. This first project may not work right the first time, so it should not be mission-critical for the company. It should be simple, self-contained, and instructive; this means that it should involve creating classes that will be meaningful to the other programmers in the company when they get their turn to learn C# and .NET.

3. Model from success

Seek out examples of good object-oriented design before starting from scratch. There's a good probability that someone has solved your problem already, and if they haven't solved it exactly you can probably apply what you've learned about abstraction to modify an existing design to fit your needs. This is the general concept of *design patterns,* covered in *Thinking in Patterns with Java*, downloadable at *www.BruceEckel.com*.

4. Use existing class libraries

The primary economic motivation for switching to OOP is the easy use of existing code in the form of class libraries (in particular, the .NET Framework SDK libraries, which are covered throughout this book). The shortest application development cycle will result when you can create and use objects from off-the-shelf libraries. However, some new programmers don't understand this, are unaware of existing class libraries, or, through fascination with the language, desire to write classes that may already exist. Your success with OOP, .NET, and C# will be optimized if you make an effort to seek out and reuse other people's code early in the transition process.

5. Don't rewrite existing code in C#

It is almost always a mistake to rewrite existing, functional code. There are incremental benefits, especially if the code is slated for reuse. But chances are you aren't going to see the dramatic increases in productivity that you hope for in your first few projects unless that project is a new one. C# and .NET shine best when taking a project from concept to reality. If you must integrate with existing code, use COM Interop or PInvoke (both discussed in Chapter 15) or, if you need even more control, write bridging code in Managed C++.

Management obstacles

If you're a manager, your job is to acquire resources for your team, to overcome barriers to your team's success, and in general to try to provide the most productive and enjoyable environment so your team is most likely to perform those miracles that are always being asked of you. Moving to .NET has benefits in all three of these categories, and it would be wonderful if it didn't cost you

anything as well. Although moving to .NET should ultimately provide a significant return on investment, it isn't free.

The most significant challenge when moving to any new language or API is the inevitable drop in productivity while new lessons are learned and absorbed. C# is no different. The everyday syntax of the C# language does not take a great deal of time to understand; a programmer familiar with a procedural programming language should be able to write simple mathematical routines by the end of a day of study. The .NET Framework SDK contains hundreds of namespaces and thousands of classes, but is well-structured and architected: this book should be adequate to guide most programmers through the most common features of the most important namespaces and give readers the knowledge required to rapidly discover additional capabilities in these areas.

The mindset of object-oriented programming, on the other hand, usually takes several months to "kick in," even when the learner is regularly exposed to good OOP code. This is not to say that the programmer cannot be productive before this, but the benefits associated with OOP (ease of testing, reuse, and maintenance) usually will not begin to accrue for several months at best. Worse, if the programmer does not have an experienced OOP programmer as a mentor, their OOP skills will often plateau very early, well before their potential is reached. The real difficulty of this situation is that the new OOP programmer will not *know* that they have fallen short of the mark they could have achieved.

Return on investment

C# and the .NET Framework have significant benefits, both direct and in the area of risk management, that should provide a significant return on investment within a year. However, as these are new technologies and because business software productivity is an area of maddeningly little concrete research, ROI calculations must be made on a company-by-company or even person-by-person basis and necessarily involve significant assumptions.

The return you will get on your investment will be in the form of software productivity: your team will be able to deliver more user value in a given period of time. But no programming language, development tool, or methodology can change a bad team into a good team. For all the fuss about everything else, software productivity can be broken down into two factors: team productivity and individual productivity.

Team productivity is always limited by communication and coordination overhead. The amount of interdependence between team members working on a single module is proportional to the square of the team size (the actual value is

(N^2-N) / 2). As Figure 1-1 illustrates, a team of 3 has just 3 avenues of communication, a team of 4 has 6 such avenues, and a team of 8 has a whopping 28 channels of communication.

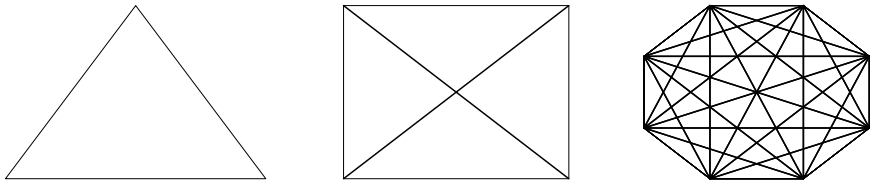


Figure 1-1 Communication paths increase faster than nodes

By the time you get beyond a small handful of programmers, this communication overhead becomes such a burden that progress becomes deadlocked. Almost all software project managers recognize this (or at least acknowledge it) and attempt to divide the project tasks into mutually independent tasks, for instance, breaking an 8-person team into two independent 4-person teams as shown in Figure 1-2.

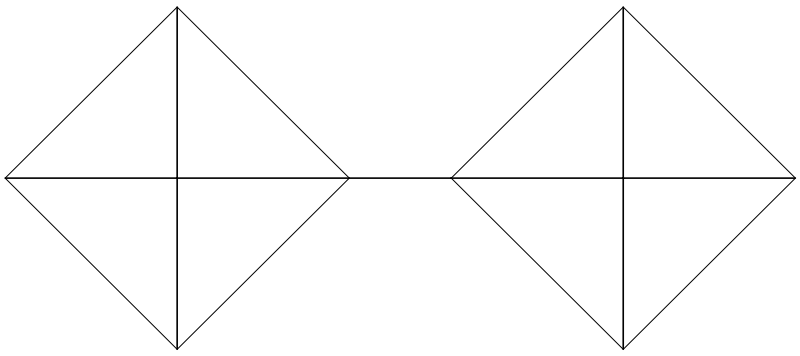


Figure 1-2 A single point of communication to each sub-team

This is a nice idea, but is difficult to achieve in practice, because it requires that each small group has a person talented enough to communicate and coordinate all their group's needs and mediate all incoming requests for information. More commonly, the best that can be accomplished is that people with such talents end up as "firewalls" for the group working on the most critical piece of functionality:

Error! Objects cannot be created from editing field codes.

Figure 1-3 "Gurus" often become supernodes within a large team

But that is not a particularly scalable solution. In the real world, the incremental benefit of adding programmers to a team diminishes rapidly even in the best-managed organization.

Individual productivity, on the other hand, has two unusual characteristics: individual programmer productivity varies by an order of magnitude, while excellent programmers are significantly rarer than terrible programmers. The very best programmers are more than twice as productive as average programmers and ten times as productive as the worst professional programmers (and because of the communication overhead of a team, this actually *understates* the contribution of excellent programmers).

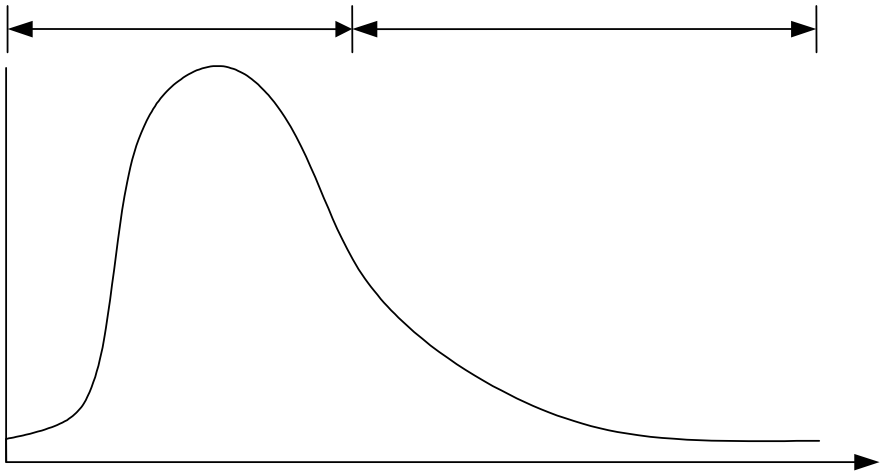


Figure 1-4 Great programmers are rarer than terrible ones

So the software management challenge is creating an efficient team of better-than-median programmers. That sounds straight from Management Platitudes 101, but it actually leads to two of the three critical questions in your ROI calculation:

- Is moving to C# going to contribute to team and individual productivity?
- Is moving to C# going to help attract and retain better-than-median programmers?

The third critical question in ROI is:

- Does moving to C# open new markets or close existing ones?

Let's take a look at each of those in turn:

Productivity comparable (but higher) than Visual Basic or Java

Given identical libraries, it would be difficult at first glance to tell a C# program from a Java one. The languages have highly similar syntaxes and C# and Java solutions to a given problem are likely to have highly similar structures.

The two major Java language features missing from C# are *inner classes* and *checked exceptions*. The primary use of Java's inner classes is to handle events, for which C# has **delegate** types; pragmatically, neither of these is a significant contributor to overall productivity. Similarly, checked exceptions are a minor burden to productivity, although some say they make a large contribution to software quality (later, we'll argue that checked exceptions do not make a great contribution to quality).

The only other significant non-library facility in Java is the object model of Enterprise JavaBeans. The four types of EJBs (stateless and stateful session beans, entity beans, and message-driven beans) provide system-level support for four needs common to enterprise systems: stateful and stateless synchronous calls, persistence, and asynchronous message-handling. While .NET provides support for all these needs, it does so in a more direct way than J2EE. J2EE introduces significant steps for distinguishing between remote and home interfaces, generating implementations, and locating, instantiating, and "narrowing" remote interfaces. While some of these steps are done only once and therefore have little long-term effect on productivity, generating EJB implementations can very significantly slow the build process from seconds to minutes, undercutting one of Java's significant value propositions. As far as enterprise development goes, C# has a significant advantage at the compiler level.

In the broad spectrum of programming languages, though, the similarities between C# and Java are far greater than their differences and their language-level productivities are certainly very close. Of course, there's far more to productivity than language-level concerns, and Java and C# do *not* share identical libraries. One can expect to see significant productivity differences based on the scope and quality of libraries. Here, one must balance the broad number of Java libraries available commercially and for free download on the Internet with the slew of COM components available to C# programmers. One can download a complete mailserver in Java or one can use COM Interop to program Outlook; which has "higher productivity" is a function of the programming task at hand. In general, though, C# appears to be poised to

challenge Java in the marketplace as the most productive language for teams building significant business applications.

One of us (Larry) has had significant experience participating and leading teams of Java developers in corporate environments, developing software for both internal and external consumption. It is Larry's belief that C# and .NET provide across-the-board productivity advantages over Java, especially when compared to J2EE and J2ME.

The development of high-quality reusable components (components approaching zero-defect level) is the most significant contributor to software productivity other than individual talent, while the development of low-quality reusable components is the most significant *detractor* from productivity (Jones, 2000). C# does not, by itself, guarantee the creation of high-quality reusables, but it does facilitate all of the key ingredients, which includes a heavy emphasis on the software engineering principles of *high cohesion* and *low coupling*. The most important quality, a fanatical devotion to defect detection and removal, is an organizational challenge.

Visual Basic has traditionally been the highest productivity environment for the rapid development of smaller Windows programs. Visual Basic facilitates the creation of programs whose internal structure mirrors the graphical structure of the program; the visual forms comprising the interface are associated with the coding for program logic. Certainly it's been possible to break away from this structure, but a decade of experience with Visual Basic has proved that a great deal of value can be delivered with a programming language that isn't overly concerned with emphasizing "computer science-y" topics but rather emphasizes the shortest cycle between thought, code, and prototype.

As programs grow in size, the proportion of effort devoted to the user interface tends to decrease and the previously mentioned issues of reuse, coupling, and cohesion control productivity. Visual Basic's productivity advantage compared to fully object-oriented languages has disappeared in larger programs. Now, with Visual Basic .NET's full support for object-orientation, that's no longer an issue. Early reports of .NET uptake show VB.NET and C# "splitting the field" in terms of adoption. While VB.NET will certainly be successful, it is more verbose than the C-derived syntax of C#. Since the languages have similar capabilities (sharing, as they do, the Common Language Infrastructure, and neither having dramatic extensions to the object-oriented imperative programming model), a C# programmer will likely be able produce equivalent functionality in fewer lines of code than a Visual Basic .NET programmer. Since the rate of lines of code produced is fairly constant across programming languages (although that rate

varies greatly between individuals), C# should have higher productivity than VB.NET.

Good, but not great, performance

C# code is compiled to a Common Intermediate Language (CIL) which is further transformed into machine-code at load-time. This *just-in-time compilation* model leads to code that runs without interpretation, but introduces two inefficiencies: an expensive loading procedure and an inability of the programmer to exploit processor knowledge. Neither of these is significant for most programmers, although elite device-driver and game programmers will likely stick with their C and C++ compilers. On the other hand, the just-in-time model provides – at least in theory – an opportunity for the JIT to produce processor-specific code that can run faster than general code. There are also interesting opportunities for profile-guided optimization.

More significantly for performance, C# uses a managed heap and a managed threading model that greatly reduce defects at the potential cost of some performance (because the runtime must run code to solve the general problem, while an elite C programmer would be able to develop a solution fine-tuned to the specifics of the task at hand). The very significant reduction in tasks associated with memory management contributes to C#'s productivity, while its performance remains acceptable for the vast majority of applications. Interestingly, C# has two characteristics (rectangular arrays and the ability to turn off array range checking) that have the *potential* for significantly increasing number-crunching speed, but casual benchmarking shows C# to remain very comparable to Java for these types of calculations. At the time of writing, rectangular arrays actually run slightly *slower* than jagged arrays, apparently because jagged array optimizations have already been implemented in the just-in-time compiler.

Low tool cost

It is possible to develop in C# with command-line tools that Microsoft makes available for free (in fact, this book advocates that programmers learn C# using these free tools in order to avoid confusing the language and its libraries from the facilities of Microsoft's Visual Studio .NET programming environment). Microsoft's Internet Information Server Webserver is bundled with their professional operating systems. One can use Microsoft Access to learn database programming with ADO.NET. A subscription to MSDN Universal, which provides the entire gamut of Microsoft development tools and servers, costs less than $3,000, which is approximately the fully-burdened cost of a programmer for one week.

The new new thing

Part of the psychology of programming is a desire to work with what is perceived as "the latest technology." The flip side of this coin is a fear of having one's skills become obsolescent, a reasonable fear in an industry that routinely undergoes huge transformations in "essential skills" every 5-6 years and has a significant prejudice against hiring older workers; David Packard's warning that "to remain static is to lose ground," has been taken to heart by generations of computer programmers. C# is the last best chance for procedural programmers to move to object orientation, while .NET provides an infrastructure that is flexible enough to embrace multiple programming paradigms as they emerge. So even if C# and .NET were only "just" as good as alternative existing languages and platforms, the best programmers are going to be attracted to opportunities to investigate these new Microsoft technologies.

A challenge for .NET, though, is the large population of second-tier programmers who may be convinced by politics or marketing not to give .NET a chance. To win the hearts and minds of the programming community, Microsoft must forego soundbite attacks and make the case that .NET is a tent big enough to hold closed and open source, individual and team development, and pragmatic and experimental programming techniques.

Access to new platforms

As discussed previously, the .NET strategy involves many more platforms beyond the desktop PC. The .NET Framework SDK directly contains capabilities appropriate for server development, while the .NET Compact Framework SDK makes programming for handhelds and other devices similarly easy. DirectX 9 will contain .NET-programmable libraries, while the TabletPC's unique features can also be accessed by C#. In addition to Microsoft's efforts to move .NET onto new platforms, the Mono project (www.go-mono.com) has brought C# to Linux.

Summary

The people who should be programmers are those who would program whether it was a profession or not. The fact is, though, that it is not just a profession, but one that plays an increasingly important role in the economy. Being a *professional* computer programmer involves understanding the economic role of information, computers, programmers, and software development "in the large." Unfortunately, an understanding of software development economics is not widespread in the business world and to be honest, neither is it widespread in the programming world itself. So a lot of effort is wasted in wild goose chases, fads, and exercises in posterior covering.

C# and the .NET Framework are the products of several underlying trends. The cost of available processing power relative to the labor cost of programming has been decreasing since the invention of computers. Into the 1970s, programmers had to compete for access to every clock cycle. This gave birth to the classic approaches to programming, both in terms of technology and, even more significantly, in terms of programmer psychology. Even in those days labor issues often drove project costs, but today, time and labor are by far the chief determinants of what can and cannot be programmed.

In the 1990s, the increasing power and interconnectedness of the machines on which software was developed and delivered combined to create significant macroeconomic effects. Even though one of these effects was a speculative bubble, other effects included real advances in the productivity across broad sectors of the economy and the rise of a new channel for delivering business value. The majority of business programming effort for the foreseeable future will be involved with delivering value via the Internet.

Analysis and design have also shifted in response to these factors. Analysis, the process of discovering the problem, and high-level design, the plan for solving the problem, are significant challenges for larger software systems. But recently, the tide of public opinion has held that the best way to solve these and other challenges of large-scale development are best handled by tackling them as a series of small projects, delivering value incrementally.

This fits with many studies of software productivity, which show that iterative development, an emphasis on quality assurance, and attention to program structure are important contributors to software success.

The C# programming language and the .NET Framework are ideally suited for the new realities of software development, but moving to C#, especially for programmers without a background in object orientation, is not without costs. Basically, object orientation does not pay off immediately or even on the first project. A new way of thinking about software and design needs to be internalized by the programmers; a good software manager will recognize that a positive return-on-investment requires an investment.

2: Introduction to Objects

This chapter is background and supplementary material. Many people do not feel comfortable wading into object-oriented programming without understanding the big picture first.

Thus, there are many concepts that are introduced here to give you a solid overview of OOP. However, many other people don't get the big picture concepts until they've seen some of the mechanics first; these people may become bogged down and lost without some code to get their hands on. If you're part of this latter group and are eager to get to the specifics of the language, feel free to jump past this chapter—skipping it at this point will not prevent you from writing programs or learning the language. However, you will want to come back here eventually to fill in your knowledge so you can understand why objects are important and how to design with them.

We'll go into great detail on the specifics of object-orientation in the first half of this book, but this chapter will introduce you to the basic concepts of OOP, including an overview of development methods. This chapter, and this book, assume that you have had experience in a procedural programming language, although not necessarily Visual Basic.

The progress of abstraction

All programming languages provide abstractions. Since to a computer everything but chip operations, register contents, and storage is an abstraction (even input and output are "just" side-effects associated with reading or writing values into particular locations), the ease with which abstractions are created and manipulated is quite important! It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. By "kind" we mean, "What is it that you are abstracting?" Assembly language is a small abstraction of the underlying machine. The early imperative

languages that followed (such as Fortran, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (in the "solution space," which is the place where you're modeling that problem, such as a computer) and the model of the problem that is actually being solved (in the "problem space," which is the place where the problem exists). The effort required to perform this mapping, and the fact that it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain, and as a side effect created the entire "programming methods" industry.

The alternative to modeling the machine is to model the problem you're trying to solve. Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic," respectively). PROLOG casts all problems as chains of true-or-false statements. Languages have been created for constraint-based programming and for programming exclusively by manipulating graphical symbols. Each of these approaches is a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects." (Of course, you will also need other objects that don't have problem-space analogs.) The idea is that the program is allowed to adapt itself to the lingo of the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction than what we've had before. Thus, OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run. There's still a connection back to the computer, though. Each object looks quite a bit like a little computer; it has a state, and it has operations that you can ask it to perform. However, this doesn't seem like such a bad analogy to objects in the real world—they all have characteristics and behaviors.

Alan Kay summarized five basic characteristics of Smalltalk, the first successful object-oriented language and one of the languages upon which C# is based. These characteristics represent a pure approach to object-oriented programming:

1. **Everything is an object.** Think of an object as a fancy variable; it stores data, but you can "make requests" to that object, asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.
2. **A program is a bunch of objects telling each other what to do by sending messages**. To make a request of an object, you "send a message" to that object. More concretely, you can think of a message as a request to call a function that belongs to a particular object.
3. **Each object has its own memory made up of other objects**. Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity in a program while hiding it behind the simplicity of objects.
4. **Every object has a type**. Using the parlance, each object is an *instance* of a *class*, in which "class" is synonymous with "type." The most important distinguishing characteristic of a class is "What messages can you send to it?"
5. **All objects of a particular type can receive the same messages**. This is actually a loaded statement, as you will see later. Because an object of type "circle" is also an object of type "shape," a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handle anything that fits the description of a shape. This *substitutability* is one of the most powerful concepts in OOP.

Booch offers an even more succinct description of an object:

An object has state, behavior and identity

This means that an object can have internal data (which gives it state), methods (to produce behavior), and each object can be uniquely distinguished from every other object – to put this in a concrete sense, each object has a unique address in memory[1].

---

[1] This is actually a bit restrictive, since objects can conceivably exist in different machines and address spaces, and they can also be stored on disk. In these cases, the identity of the object must be determined by something other than memory address, for instance, a Uniform Resource Indicator (URI).

An object has an interface

Aristotle was probably the first to begin a careful study of the concept of *type;* he spoke of "the class of fishes and the class of birds." The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword **class** that introduces a new type into a program.

Simula, as its name implies, was created for developing simulations such as the classic "bank teller problem." In this, you have a bunch of tellers, customers, accounts, transactions, and units of money—a lot of "objects." Objects that are identical except for their state during a program's execution are grouped together into "classes of objects" and that's where the keyword **class** came from. Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: You can create variables of a type (called *objects* or *instances* in object-oriented parlance) and manipulate those variables (called *sending messages* or *requests;* you send a message and the object figures out what to do with it). The members (elements) of each class share some commonality: every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state, each account has a different balance, each teller has a name. Thus, the tellers, customers, accounts, transactions, etc., can each be represented with a unique entity in the computer program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors.

So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "class" keyword. C# has some data types that are not classes, but in general, when you see the word "type" think "class" and vice versa[2].

Since a class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs. The programming system

---

[2] Some people make a distinction, stating that type determines the interface while class is a particular implementation of that interface.

welcomes the new classes and gives them all the care and type-checking that it gives to built-in types.

The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you're designing, the use of OOP techniques can easily reduce a large set of problems to a simple solution.

Once a class is established, you can make as many objects of that class as you like, and then manipulate those objects as if they are the elements that exist in the problem you are trying to solve. Indeed, one of the challenges of object-oriented programming is to create a one-to-one mapping between the elements in the problem space and objects in the solution space.

But how do you get an object to do useful work for you? There must be a way to make a request of the object so that it will do something, such as complete a transaction, draw something on the screen, or turn on a switch. And each object can satisfy only certain requests. The requests you can make of an object are defined by its interface, and the type is what determines the interface. A simple example might be a representation of a light bulb:

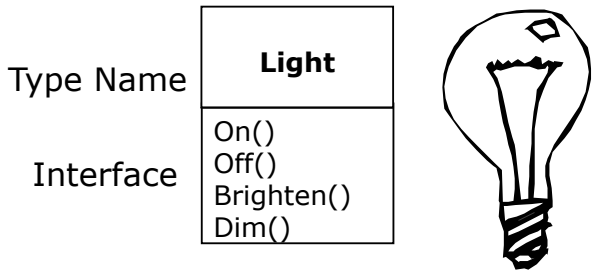


Figure 2-1 An object has an interface

```
Light lt = new Light();
lt.On();
```

The interface establishes *what* requests you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the *implementation*. From a procedural programming standpoint, it's not that complicated. A type has a function associated with each possible request, and when you make a particular request to an object, that function is called. This process is usually summarized by saying

that you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code).

Here, the name of the type/class is **Light**, the name of this particular **Light** object is **lt**, and the requests that you can make of a **Light** object are to turn it on, turn it off, make it brighter, or make it dimmer. You create a **Light** object by defining a "reference" (**lt**) for that object and calling **new** to request a new object of that type. To send a message to the object, you state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a predefined class, that's pretty much all there is to programming with objects.

The diagram shown above follows the format of the Unified Modeling Language (UML). Each class is represented by a box, with the type name in the top portion of the box, any data members that you care to describe in the middle portion of the box, and the *member functions* (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box. Often, only the name of the class and the public member functions are shown in UML design diagrams, and so the middle portion is not shown. If you're interested only in the class name, then the bottom portion doesn't need to be shown, either.

This book will gradually present more and more UML diagrams of different types, introducing them as appropriate for specific needs. As was mentioned earlier, the UML is a language *at least* as complicated as C# itself, but *Thinking in UML* would be a very different book from this one[3]. The diagrams in this book do not always comply with the letter of the UML specification and are drawn with the sole goal of clarifying the main text.

The hidden implementation

It is helpful to break up the playing field into *class creators* (those who create new data types) and *client programmers* (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't use it, which means that the class creator can change the hidden portion at will without worrying about the impact on anyone else. The

---

[3] *Thinking in UML* doesn't exist!

hidden portion usually represents the tender insides of an object that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs. The concept of implementation hiding cannot be overemphasized.

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is also a programmer, but one who is putting together an application by using your library, possibly to build a bigger library.

If all the members of a class are available to everyone, then the client programmer can do anything with that class and there's no way to enforce rules. Even though you might really prefer that the client programmer not directly manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

So the first reason for access control is to keep client programmers' hands off portions they shouldn't touch—parts that are necessary for the internal machinery of the data type but not part of the interface that users need in order to solve their particular problems. This is actually a service to users because they can easily see what's important to them and what they can ignore.

The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. For example, you might implement a particular class in a simple fashion to ease development, and then later discover that you need to rewrite it in order to make it run faster. If the interface and implementation are clearly separated and protected, you can accomplish this easily.

C# uses five explicit keywords to set the boundaries in a class: **public**, **private**, **protected**, **internal,** and **protected internal**. Their use and meaning are quite straightforward. These *access specifiers* determine who can use the definitions that follow. **public** means the following definitions are available to everyone. The **private** keyword, on the other hand, means that no one can access those definitions except you, the creator of the type, inside member functions of that type. **private** is a brick wall between you and the client programmer. If someone tries to access a **private** member, they'll get a compile-time error. **protected** acts like **private**, with the exception that an inheriting class has access to **protected** members, but not **private** members. Inheritance will be introduced shortly. **internal** is often called "friendly"–the definition can be accessed by other classes in the same *assembly* (a DLL or EXE file used to distribute .NET classes) as if it were **public**, but is not accessible to classes in different assemblies. **protected internal** allows access by classes within the

same assembly (as with **internal**) *or* by inheriting classes (as with **protected**) even if the inheriting classes are not within the same assembly.

C#'s default access, which comes into play if you don't use one of the aforementioned specifiers, is **internal** for classes and **private** for class members.

Reusing the implementation

Once a class has been created and tested, it should (ideally) represent a useful unit of code. It turns out that this reusability is not nearly so easy to achieve as many would hope; it takes experience and insight to produce a good design. But once you have such a design, it begs to be reused. Code reuse is one of the greatest advantages that object-oriented programming languages provide.

The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class. We call this "creating a member object." Your new class can be made up of any number and type of other objects, in any combination that you need to achieve the functionality desired in your new class. Because you are composing a new class from existing classes, this concept is called *composition* (or more generally, *aggregation*). Composition is often referred to as a "has-a" relationship, as in "a car has an engine."

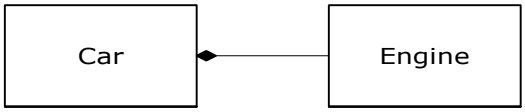


Figure 2-2: A Car *has an* Engine

(The above UML diagram indicates composition with the filled diamond, which states the Engine is contained within the car. We will typically use a simpler form: just a line, without the diamond, to indicate an association.[4])

Composition comes with a great deal of flexibility. The member objects of your new class are usually private, making them inaccessible to the client programmers who are using the class. This allows you to change those members without disturbing existing client code. You can also change the member objects at run-time, to dynamically change the behavior of your program. Inheritance,

[4] This is usually enough detail for most diagrams, and you don't need to get specific about whether you're using aggregation or composition.

which is described next, does not have this flexibility since the compiler must place compile-time restrictions on classes created with inheritance.

Because inheritance is so important in object-oriented programming it is often highly emphasized, and the new programmer can get the idea that inheritance should be used everywhere. This can result in awkward and overly complicated designs. Instead, you should first look to composition when creating new classes, since it is simpler and more flexible. If you take this approach, your designs will be cleaner. Once you've had some experience, it will be reasonably obvious when you need inheritance.

Inheritance: Reusing the interface

By itself, the idea of an object is a convenient tool. It allows you to package data and functionality together by *concept*, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the **class** keyword.

It seems a pity, however, to go to all the trouble to create a class and then be forced to create a brand new one that might have similar functionality. It's nicer if we can take the existing class, clone it, and then make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (called the *base* or *super* or *parent* class) is changed, the modified "clone" (called the *derived* or *inherited* or *sub* or *child* class) also reflects those changes.

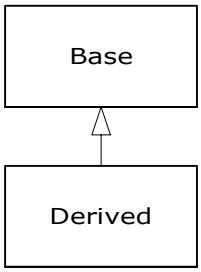


Figure 2-3: Derived *is a type of* Base

(The arrow in the above UML diagram points from the derived class to the base class. As you will see, there can be more than one derived class.)

A type does more than describe the constraints on a set of objects; it also has a relationship with other types. Two types can have characteristics and behaviors in common, but one type may contain more characteristics than another and may also handle more messages (or handle them differently). Inheritance expresses this similarity between types using the concept of base types and derived types. A base type contains all of the characteristics and behaviors that are shared among the types derived from it. You create a base type to represent the core of your ideas about some objects in your system. From the base type, you derive other types to express the different ways that this core can be realized.

For example, a trash-recycling machine sorts pieces of trash. The base type is "trash," and each piece of trash has a weight, a value, and so on, and can be shredded, melted, or decomposed. From this, more specific types of trash are derived that may have additional characteristics (a bottle has a color) or behaviors (an aluminum can may be crushed, a steel can is magnetic). In addition, some behaviors may be different (the value of paper depends on its type and condition). Using inheritance, you can build a type hierarchy that expresses the problem you're trying to solve in terms of its types.

A second example is the classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc. From this, specific types of shapes are derived (inherited): circle, square, triangle, and so on, each of which may have additional characteristics and behaviors. Certain shapes can be flipped, for example. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.

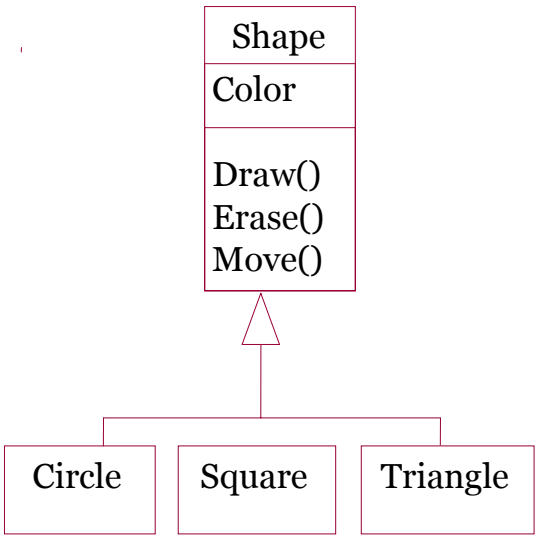


Figure 2-4: All subtypes share the same behavior names

Casting the solution in the same terms as the problem is tremendously beneficial because you don't need a lot of intermediate models to get from a description of the problem to a description of the solution. With objects, the type hierarchy is the primary model, so you go directly from the description of the system in the real world to the description of the system in code. Indeed, one of the difficulties people have with object-oriented design is that it's too simple to get from the beginning to the end. A mind trained to look for complex solutions is often stumped by this simplicity at first.

When you inherit from an existing type, you create a new type. This new type contains not only all the members of the existing type (although the **private** ones are hidden away and inaccessible), but more important, it duplicates the interface of the base class. That is, all the messages you can send to objects of the base class you can also send to objects of the derived class. Since we know the type of a class by the messages we can send to it, this means that the derived class *is the same type as the base class*. In the previous example, "a circle is a shape." This type equivalence via inheritance is one of the fundamental gateways in understanding the meaning of object-oriented programming.

Since both the base class and derived class have the same interface, there must be some implementation to go along with that interface. That is, there must be some code to execute when an object receives a particular message. If you simply inherit a class and don't do anything else, the methods from the base-class

interface come right along into the derived class. That means objects of the derived class have not only the same type, they also have the same behavior, which isn't particularly interesting.

You have two ways to differentiate your new derived class from the original base class. The first is quite straightforward: You simply add brand new functions to the derived class. These new functions are not part of the base class interface. This means that the base class simply didn't do as much as you wanted it to, so you added more functions. This simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, you should look closely for the possibility that your base class might also need these additional functions. This process of discovery and iteration of your design happens regularly in object-oriented programming.

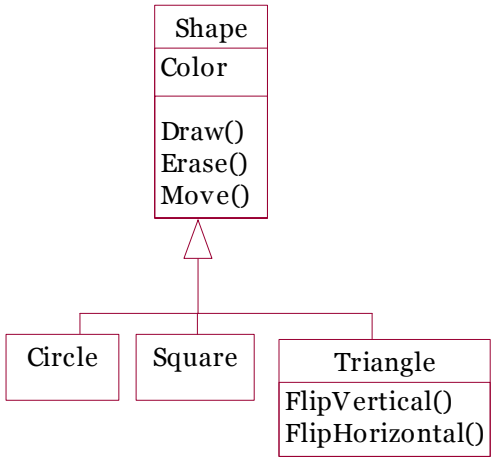


Figure 2-5: Derived classes *may* extend the base interface

Although inheritance may sometimes imply that you are going to add new functions to the interface, that's not necessarily true. The second and more important way to differentiate your new class is to *change* the behavior of an existing base-class function. This is referred to as *overriding* that function.

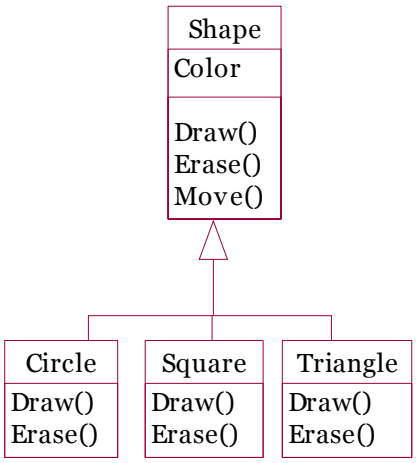


Figure 2-6: The *name* of the behavior is constant, the behavior itself may vary

To override a function, you simply create a new definition for the function in the derived class. You're saying, "I'm using the same interface function here, but I want it to do something different for my new type."

Is-a vs. is-like-a relationships

There's a certain debate that can occur about inheritance: Should inheritance override *only* base-class functions (and not add new member functions that aren't in the base class)? This would mean that the derived type is *exactly* the same type as the base class since it has exactly the same interface. As a result, you can exactly substitute an object of the derived class for an object of the base class. This can be thought of as *pure substitution*, and it's often referred to as the *substitution principle*. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this case as an *is-a* relationship, because you can say "a circle *is a* shape." A test for inheritance is to determine whether you can state the is-a relationship about the classes and have it make sense.

There are times when you must add new interface elements to a derived type, thus extending the interface and creating a new type. The new type can still be substituted for the base type, but the substitution isn't perfect because your new functions are not accessible from the base type. This can be described as an *is-like-a* relationship; the new type has the interface of the old type but it also contains other functions, so you can't really say it's exactly the same. For example, consider an air conditioner. Suppose your house is wired with all the controls for cooling; that is, it has an interface that allows you to control cooling.

Imagine that the air conditioner breaks down and you replace it with a heat pump, which can both heat and cool. The heat pump *is-like-an* air conditioner, but it can do more. Because the control system of your house is designed only to control cooling, it is restricted to communication with the cooling part of the new object. The interface of the new object has been extended, and the existing system doesn't know about anything except the original interface.

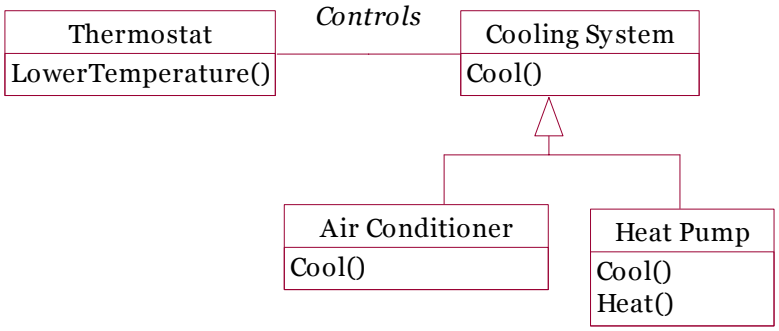


Figure 2-7: HeatPump *is like a* CoolingSystem, but is this the best solution?

Of course, once you see this design it becomes clear that the base class "cooling system" is not general enough, and should be renamed to "temperature control system" so that it can also include heating—at which point the substitution principle will work. However, the diagram above is an example of what can happen in design and in the real world.

When you see the substitution principle it's easy to feel like this approach (pure substitution) is the only way to do things, and in fact it is nice if your design works out that way. But you'll find that there are times when it's equally clear that you must add new functions to the interface of a derived class. With inspection both cases should be reasonably obvious.

Interchangeable objects with polymorphism

When dealing with type hierarchies, you often want to treat an object not as the specific type that it is, but instead as its base type. This allows you to write code that doesn't depend on specific types. In the shape example, functions manipulate generic shapes without respect to whether they're circles, squares, triangles, or some shape that hasn't even been defined yet. All shapes can be drawn, erased, and moved, so these functions simply send a message to a shape object; they don't worry about how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is the most common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called pentagon without modifying the functions that deal only with generic shapes. This ability to extend a program easily by deriving new subtypes is important because it greatly improves designs while reducing the cost of software maintenance.

There's a problem, however, with attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a function is going to tell a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to move, the compiler cannot know at compile-time precisely what piece of code will be executed. That's the whole point—when the message is sent, the programmer doesn't want to know what piece of code will be executed; the draw function can be applied equally to a circle, a square, or a triangle, and the object will execute the proper code depending on its specific type. If you don't have to know what piece of code will be executed, then when you add a new subtype, the code it executes can be different without requiring changes to the function call. Therefore, the compiler cannot know precisely what piece of code is executed, so what does it do? For example, in the following diagram the **BirdController** object just works with generic **Bird** objects, and does not know what exact type they are. This is convenient from **BirdController**'s perspective because it doesn't have to write special code to determine the exact type of **Bird** it's working with, or that **Bird**'s behavior. So how does it happen that, when **Move( )** is called while ignoring the specific type of **Bird**, the right behavior will occur (a **Goose** runs, flies, or swims, and a **Penguin** runs or swims)?

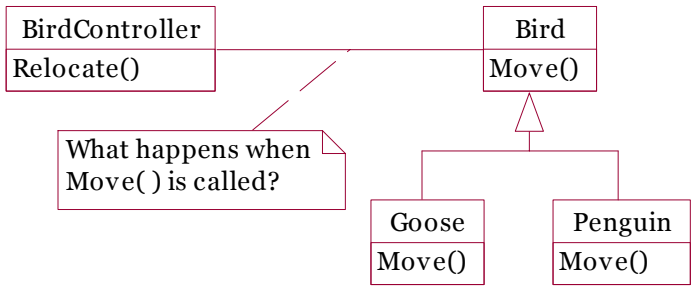


Figure 2-8: Late binding is the primary twist in OOP

The answer is the primary twist in object-oriented programming: the language does not make a function call in the traditional sense. The function call generated by a non-OOP language causes what is called *early binding*, a term you may not have heard before because you've never thought about it any other way. It means

the compiler generates a call to a specific function name, and the linker resolves this call to the absolute address of the code to be executed. In OOP, the program cannot determine the address of the code until run-time, so some other scheme is necessary when a message is sent to a generic object.

To solve the problem, object-oriented languages use the concept of *late binding*. When you send a message to an object, the code being called isn't determined until run-time. The compiler does ensure that the function exists and performs type checking on the arguments and return value (languages such as Visual Basic in which this isn't true are said to have *weak typing* or *latent typing*), but it doesn't know the exact code to execute.

To perform late binding, C# uses a special bit of code in lieu of the absolute call. This code calculates the address of the function body, using information stored in the object (this process is covered in great detail in Chapter 7). Thus, each object can behave differently according to the contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

In C#, you can choose whether a language method call is early- or late-bound. By default, they are early-bound. To take advantage of polymorphism, methods must be defined in the base class using the **virtual** keyword and implemented in inheriting classes with the **override** keyword.

Consider the shape example. The family of classes (all based on the same uniform interface) was diagrammed earlier in this chapter. To demonstrate polymorphism, we want to write a single piece of code that ignores the specific details of type and talks only to the base class. That code is *decoupled* from type-specific information, and thus is simpler to write and easier to understand. And, if a new type—a **Hexagon**, for example—is added through inheritance, the code you write will work just as well for the new type of **Shape** as it did on the existing types. Thus, the program is *extensible*.

If you write a method in C# (as you will soon learn how to do):

```
void DoStuff(Shape s) {
  s.Erase();
  // ...
  s.Draw();
}
```

This function speaks to any **Shape**, so it is independent of the specific type of object that it's drawing and erasing. If in some other part of the program we use the **DoStuff( )** function:

```
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
DoStuff(c);
DoStuff(t);
DoStuff(l);
```

The calls to **DoStuff( )** automatically work correctly, regardless of the exact type of the object.

This is actually a pretty amazing trick. Consider the line:

```
DoStuff(c);
```

What's happening here is that a **Circle** is being passed into a function that's expecting a **Shape**. Since a **Circle** *is* a **Shape** it can be treated as one by **DoStuff( )**. That is, any message that **DoStuff( )** can send to a **Shape**, a **Circle** can accept. So it is a completely safe and logical thing to do.

We call this process of treating a derived type as though it were its base type *upcasting*. The name *cast* is used in the sense of casting into a mold and the *up* comes from the way the inheritance diagram is typically arranged, with the base type at the top and the derived classes fanning out downward. Thus, casting to a base type is moving up the inheritance diagram: "upcasting."

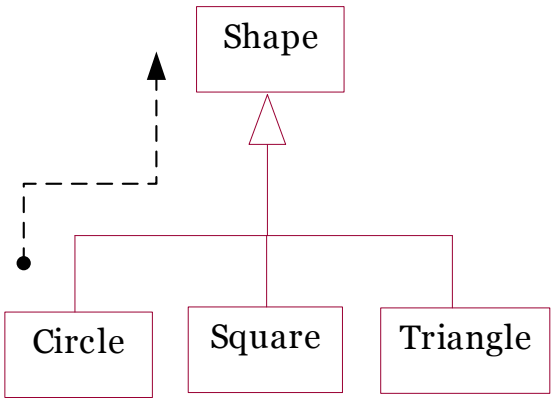


Figure 2-9: A subtype can be *upcast* to its base type(s)

An object-oriented program contains some upcasting somewhere, because that's how you decouple yourself from knowing about the exact type you're working with. Look at the code in **DoStuff( )**:

```
s.Erase();
// ...
s.Draw();
```

Notice that it doesn't say "If you're a **Circle**, do this, if you're a **Square**, do that, etc." If you write that kind of code, which checks for all the possible types that a **Shape** can actually be, it's messy and you need to change it every time you add a new kind of **Shape**. Here, you just say "You're a shape, I know you can **Erase( )** and **Draw( )** yourself, do it, and take care of the details correctly."

What's impressive about the code in **DoStuff( )** is that, somehow, the right thing happens. Calling **Draw( )** for **Circle** causes different code to be executed than when calling **Draw( )** for a **Square** or a **Line**, but when the **Draw( )** message is sent to an anonymous **Shape**, the correct behavior occurs based on the actual type of the **Shape**. This is amazing because, as mentioned earlier, when the C# compiler is compiling the code for **DoStuff( )**, it cannot know exactly what types it is dealing with. So ordinarily, you'd expect it to end up calling the version of **Erase( )** and **Draw( )** for the base class **Shape**, and not for the specific **Circle**, **Square**, or **Line**. And yet the right thing happens because of polymorphism. The compiler and run-time system handle the details; all you need to know is that it happens, and more important how to design with it. When you send a message to an object, the object will do the right thing, even when upcasting is involved.

Abstract base classes and interfaces

Often in a design, you want the base class to present *only* an interface for its derived classes. That is, you don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by making that class *abstract* using the **abstract** keyword. If anyone tries to make an object of an **abstract** class, the compiler prevents them. This is a tool to enforce a particular design.

You can also use the **abstract** keyword to describe a method that hasn't been implemented yet—as a stub indicating "here is an interface function for all types inherited from this class, but at this point I don't have any implementation for it." An **abstract** method may be created only inside an **abstract** class. When the class is inherited, that method must be implemented, or the inheriting class becomes **abstract** as well. Creating an **abstract** method allows you to put a method in an interface without being forced to provide a possibly meaningless body of code for that method.

The **interface** keyword takes the concept of an **abstract** class one step further by preventing any function definitions at all. The **interface** is a very handy and commonly used tool, as it provides the perfect separation of interface and implementation. In addition, you can combine many interfaces together, if you wish, whereas inheriting from multiple regular classes or abstract classes is not possible.

Object landscapes and lifetimes

Technically, OOP is just about abstract data typing, inheritance, and polymorphism, but other issues can be at least as important. The remainder of this section will cover these issues.

One of the most important factors is the way objects are created and destroyed. Where is the data for an object and how is the lifetime of the object controlled? There are different philosophies at work here. C++ takes the approach that control of efficiency is the most important issue, so it gives the programmer a choice. For maximum run-time speed, the storage and lifetime can be determined while the program is being written, by placing the objects on the stack (these are sometimes called *automatic* or *scoped* variables) or in the static storage area. This places a priority on the speed of storage allocation and release, and control of these can be very valuable in some situations. However, you sacrifice flexibility because you must know the exact quantity, lifetime, and type of objects while you're writing the program. If you are trying to solve a more general problem

such as computer-aided design, warehouse management, or air-traffic control, this is too restrictive.

The second approach is to create objects dynamically in a pool of memory called the heap. In this approach, you don't know until run-time how many objects you need, what their lifetime is, or what their exact type is. Those are determined at the spur of the moment while the program is running. If you need a new object, you simply make it on the heap at the point that you need it. Because the storage is managed dynamically, at run-time, the amount of time required to allocate storage on the heap is significantly longer than the time to create storage on the stack. (Creating storage on the stack is often a single assembly instruction to move the stack pointer down, and another to move it back up.) The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object. In addition, the greater flexibility is essential to solve the general programming problem.

C# uses the second approach exclusively, except for *value types*, which will be discussed shortly. Every time you want to create an object, you use the **new** keyword to build a dynamic instance of that object. With languages that allow objects to be created on the stack, the compiler determines how long the object lasts and can automatically destroy it. However, if you create it on the heap the compiler has no knowledge of its lifetime. In a language like C++, you must determine programmatically when to destroy the object, which can lead to memory leaks if you don't do it correctly (and this is a common problem in C++ programs). The .NET runtime provides a feature called a garbage collector that automatically discovers when an object is no longer in use and destroys it. A garbage collector is much more convenient because it reduces the number of issues that you must track and the code you must write. More important, the garbage collector provides a much higher level of insurance against the insidious problem of memory leaks (which has brought many a C++ project to its knees).

The rest of this section looks at additional factors concerning object lifetimes and landscapes.

Collections and iterators

If you don't know how many objects you're going to need to solve a particular problem, or how long they will last, you also don't know how to store those objects. How can you know how much space to create for those objects? You can't, since that information isn't known until run-time.

The solution to most problems in object-oriented design seems flippant: you create another type of object. The new type of object that solves this particular problem holds references to other objects. Of course, you can do the same thing with an array, which is available in most languages. But there's more. This new object, generally called a *container* (also called a *collection*), will expand itself whenever necessary to accommodate everything you place inside it. So you don't need to know how many objects you're going to hold in a container. Just create a container object and let it take care of the details.

Fortunately, a good OOP language comes with a set of containers as part of the package. In C++, it's part of the Standard C++ Library and is sometimes called the Standard Template Library (STL). Object Pascal has containers in its Visual Component Library (VCL). Smalltalk has a very complete set of containers. Like Java, C# also has containers in its standard library. In some libraries, a generic container is considered good enough for all needs, and in others (C#, for example) the library has different types of containers for different needs: a vector (called an **ArrayList** in C#), queues, hashtables, trees, stacks, etc.

All containers have some way to put things in and get things out; there are usually functions to add elements to a container, and others to fetch those elements back out. But fetching elements can be more problematic, because a single-selection function is restrictive. What if you want to manipulate or compare a set of elements in the container instead of just accessing a single element?

The solution is an enumerator, which is an object whose job is to select the elements within a container and present them to the user of the iterator. As a class, it also provides a level of abstraction. This abstraction can be used to separate the details of the container from the code that's accessing that container. The container, via the enumerator, is abstracted to be simply a sequence. The enumerator allows you to traverse that sequence without worrying about the underlying structure—that is, whether it's an **ArrayList**, a **Hashtable**, a **Stack**, or something else. This gives you the flexibility to easily change the underlying data structure without disturbing the code in your program.

From a design standpoint, all you really want is a sequence that can be manipulated to solve your problem. If a single type of sequence satisfied all of your needs, there'd be no reason to have different kinds. There are two reasons that you need a choice of containers. First, containers provide different types of interfaces and external behavior. A stack has a different interface and behavior than that of a queue, which is different from that of a dictionary or a list. One of these might provide a more flexible solution to your problem than the other.

Second, different containers have different efficiencies for certain operations. But in the end, remember that a container is only a storage cabinet to put objects in. If that cabinet solves all of your needs, it doesn't really matter how it is implemented (a basic concept with most types of objects).

The singly rooted hierarchy

One of the issues in OOP that has become especially prominent since the introduction of C++ is whether all classes should ultimately be inherited from a single base class. In C# (as with virtually all other OOP languages) the answer is "yes" and the name of this ultimate base class is simply **object**. It turns out that the benefits of the singly rooted hierarchy are many.

All objects in a singly rooted hierarchy have an interface in common, so they are all ultimately the same type. The alternative (provided by C++) is that you don't know that everything is the same fundamental type. From a backward-compatibility standpoint this fits the model of C better and can be thought of as less restrictive, but when you want to do full-on object-oriented programming you must then build your own hierarchy to provide the same convenience that's built into other OOP languages. And in any new class library you acquire, some other incompatible interface will be used. It requires effort (and possibly multiple inheritance) to work the new interface into your design. Is the extra "flexibility" of C++ worth it? If you need it—if you have a large investment in C—it's quite valuable. If you're starting from scratch, other alternatives such as C# can often be more productive.

All objects in a singly rooted hierarchy (such as C# provides) can be guaranteed to have certain functionality. You know you can perform certain basic operations on every object in your system. A singly rooted hierarchy, along with creating all objects on the heap, greatly simplifies argument passing (one of the more complex topics in C++).

A singly rooted hierarchy makes it much easier to implement a garbage collector (which is conveniently built into C#). The necessary support can be installed in the base class, and the garbage collector can thus send the appropriate messages to every object in the system. Without a singly rooted hierarchy and a system to manipulate an object via a reference, it is difficult to implement a garbage collector.

Since C# guarantees that run-time type information is available in all objects, you'll never end up with an object whose type you cannot determine. This is especially important with system level operations, such as exception handling, and to allow greater flexibility in programming.

Collection libraries and support for easy collection use

Because a container is a tool that you'll use frequently, it makes sense to have a library of containers that are built in a reusable fashion, so you can take one off the shelf and plug it into your program. .NET provides such a library, which should satisfy most needs.

Downcasting vs. templates/generics

To make these containers reusable, they hold the one universal type in .NET that was previously mentioned: **object**. The singly rooted hierarchy means that everything is an **object**, so a container that holds **object**s can hold anything. This makes containers easy to reuse.

To use such a container, you simply add object references to it, and later ask for them back. But, since the container holds only **object**s, when you add your object reference into the container it is upcast to **object**, thus losing its identity. When you fetch it back, you get an **object** reference, and not a reference to the type that you put in. So how do you turn it back into something that has the useful interface of the object that you put into the container?

Here, the cast is used again, but this time you're not casting up the inheritance hierarchy to a more general type, you cast down the hierarchy to a more specific type. This manner of casting is called *downcasting*. With upcasting, you know, for example, that a **Circle** is a type of **Shape** so it's safe to upcast, but you don't know that an **object** is necessarily a **Circle** or a **Shape** so it's hardly safe to downcast unless you know that's what you're dealing with.

It's not completely dangerous, however, because if you downcast to the wrong thing you'll get a run-time error called an *exception,* which will be described shortly. When you fetch object references from a container, though, you must have some way to remember exactly what they are so you can perform a proper downcast.

Downcasting and the run-time checks require extra time for the running program, and extra effort from the programmer. Wouldn't it make sense to somehow create the container so that it knows the types that it holds, eliminating the need for the downcast and a possible mistake? The solution is parameterized types, which are classes that the compiler can automatically customize to work with particular types. For example, with a parameterized container, the compiler could customize that container so that it would accept only Shapes and fetch only Shapes.

Parameterized types are an important part of C++, partly because C++ has no singly rooted hierarchy. In C++, the keyword that implements parameterized types is "template." .NET currently has no parameterized types since it is possible for it to get by—however awkwardly—using the singly rooted hierarchy. However, there is no doubt that parameterized types will be implemented in a future version of the .NET Framework.

The housekeeping dilemma: who should clean up?

Each object requires resources in order to exist, most notably memory. When an object is no longer needed it must be cleaned up so that these resources are released for reuse. In simple programming situations the question of how an object is cleaned up doesn't seem too challenging: you create the object, use it for as long as it's needed, and then it should be destroyed. It's not hard, however, to encounter situations that are more complex.

Suppose, for example, you are designing a system to manage air traffic for an airport. (The same model might also work for managing crates in a warehouse, or a video rental system, or a kennel for boarding pets.) At first it seems simple: make a container to hold airplanes, then create a new airplane and place it in the container for each airplane that enters the air-traffic-control zone. For cleanup, simply delete the appropriate airplane object when a plane leaves the zone.

But perhaps you have some other system to record data about the planes, possibly data that doesn't require such immediate attention as the main controller function. Maybe it's a record of the flight plans of all the small planes that leave the airport. So you have a second container of small planes, and whenever you create a plane object you also put it in this second container if it's a small plane. Then some background process performs operations on the objects in this container during idle moments.

Now the problem is more difficult: how can you possibly know when to destroy the objects? When you're done with the object, some other part of the system might not be. This same problem can arise in a number of other situations, and in programming systems (such as C++) in which you must explicitly delete an object when you're done with it this can become quite complex.

With C#, the garbage collector is designed to take care of the problem of releasing the memory (although this doesn't include other aspects of cleaning up an object). When the garbage collector runs (which can happen at any time), it can tell which objects are no longer in use, and it automatically releases the memory for that object. This (combined with the fact that all objects are inherited from

the single root class **object** and that you can create objects only one way, on the heap) makes the process of programming in C# much simpler than programming in C++. You have far fewer decisions to make and hurdles to overcome.

Garbage collectors vs. efficiency and flexibility

If all this is such a good idea, why didn't they do the same thing in C++? Well of course there's a price you pay for all this programming convenience, and that price is run-time overhead. As mentioned before, in C++ you can create objects on the stack, and in this case they're automatically cleaned up (but you don't have the flexibility of creating as many as you want at run-time). Creating objects on the stack is the most efficient way to allocate storage for objects and to free that storage. Creating objects on the heap can be much more expensive. Always inheriting from a base class and making all function calls polymorphic also exacts a small toll. But the garbage collector is a particular problem because you never quite know when it's going to start up or how long it will take. This means that there's an inconsistency in the rate of execution of a C# program, so you can't use it in certain situations, such as when the rate of execution of a program is uniformly critical. (These are generally called real-time programs, although not all real-time programming problems are this stringent.)

The designers of the C++ language, trying to woo C programmers (and most successfully, at that), did not want to add any features to the language that would impact the speed or the use of C++ in any situation where programmers might otherwise choose C. This goal was realized, but at the price of greater complexity when programming in C++. C# is much simpler to use than C++, but the trade-off is in efficiency and sometimes applicability. For a significant portion of programming problems, however, C# is the superior choice.

Exception handling: dealing with errors

Since programming languages were created, error handling has been one of the most difficult issues. Because it's so hard to design a good error handling scheme, many languages simply ignore the issue, passing the problem on to library designers who come up with halfway measures that can work in many situations but can easily be circumvented, generally by just ignoring them. A major problem with most error handling schemes is that they rely on programmer vigilance in following an agreed-upon convention that is not enforced by the language. If the programmer is not vigilant—often the case if they are in a hurry—these schemes can easily be forgotten.

Exception handling wires error handling directly into the programming language and sometimes even the operating system. An exception is an object that is "thrown" from the site of the error and can be "caught" by an appropriate exception handler designed to handle that particular type of error. It's as if exception handling is a different, parallel path of execution that can be taken when things go wrong. And because it uses a separate execution path, it doesn't need to interfere with your normally executing code. This makes that code simpler to write since you aren't constantly forced to check for errors. In addition, a thrown exception is unlike an error value that's returned from a function or a flag that's set by a function in order to indicate an error condition—these can be ignored. An exception cannot be ignored, so it's guaranteed to be dealt with at some point. Finally, exceptions provide a way to reliably recover from a bad situation. Instead of just exiting you are often able to set things right and restore the execution of a program, which produces much more robust programs.

It's worth noting that exception handling isn't an object-oriented feature, although in object-oriented languages the exception is normally represented with an object. Exception handling existed before object-oriented languages.

Multithreading

A fundamental concept in computer programming is the idea of handling more than one task at a time. Many programming problems require that the program be able to stop what it's doing, deal with some other problem, and then return to the main process. The solution has been approached in many ways. Initially, programmers with low-level knowledge of the machine wrote interrupt service routines and the suspension of the main process was initiated through a hardware interrupt. Although this worked well, it was difficult and nonportable, so it made moving a program to a new type of machine slow and expensive.

Sometimes interrupts are necessary for handling time-critical tasks, but there's a large class of problems in which you're simply trying to partition the problem into separately running pieces so that the whole program can be more responsive, or often just simpler to create and understand. Within a program, these separately running pieces are called threads, and the general concept is called *multithreading*. A common example of multithreading is the user interface. By using threads, a user can press a button and get a quick response rather than being forced to wait until the program finishes its current task.

Ordinarily, threads are just a way to allocate the time of a single processor. But if the operating system supports multiple processors, each thread can be assigned

to a different processor and they can truly run in parallel. One of the convenient features of multithreading at the language level is that the programmer doesn't need to worry about whether there are many processors or just one. The program is logically divided into threads and if the machine has more than one processor and can allocate the hardware as a "processor pool," then the program runs faster without any special adjustments.

All this makes threading sound pretty simple. There is a catch: shared resources. If you have more than one thread running that's expecting to access the same resource you have a problem. For example, two processes can't simultaneously send information to a printer. To solve the problem, resources that can be shared, such as the printer, must be locked while they are being used. So a thread locks a resource, completes its task, and then releases the lock so that someone else can use the resource.

C#'s threading is built into the language, which makes a complicated subject much simpler. The threading is supported on an object level, so one thread of execution is represented by one object. C# also provides limited resource locking. It can lock the memory of any object (which is, after all, one kind of shared resource) so that only one thread can use it at a time. This is accomplished with the **lock** keyword. Other types of resources must be locked explicitly by the programmer, typically by creating an object to represent the lock that all threads must check before accessing that resource.

Persistence

When you create an object, it exists as long as you need it, but it ceases to exist when the program terminates. While this makes sense at first, there are situations where it would be incredibly useful if an object could be created during one program run and then be transported across program and computer boundaries or be brought back into its fully-formed existence the next time the program is run. One way of doing this is to create a database table whose columns correspond to the fields of the object and write code that maps an object's state to a single record in the database. Another way is to use XML to represent the persistent state of the object. C# has two serialization schemes; one based on a binary representation of the object and the other that uses XML. The XML scheme, while a little more work to implement than the binary one, can mediate between objects and XML documents, which in turn can be stored in files, transmitted over the Internet, or can themselves be mapped into database files.

Remote objects

While databases and serialization are fine for persisting data, an increasing number of programs are written to cooperate with other programs by the use of *remote procedure calls (RPC)*. Where persistence allows you to store and retrieve the *state* of your objects, RPC adds behavior. As Booch defined it above, an object has state, behavior, and identity, so a distributed object-oriented system needs those three things as well. .NET supports two types of distributed object-oriented programming:

- *.NET Remoting* is an efficient "native" architecture in which objects have identity inside *application domains* that may or may not be on the local machine.
- *Web Services* uses an XML-based RPC mechanism that uses familiar Universal Resource Identifiers (URIs) to define identity. Essentially, when a remote method call is made with SOAP as the format and HTTP as the medium, you've got a Web Service.While the XML-RPC is not as efficient on a call-by-call basis as binary .NET Remoting, Web Services are not confined to the .NET platform and are therefore the subject of a broader range of development efforts.

Obviously, the coordination of multiple machines and the transfer of critical data over the vast, unpredictable, and uncontrollable Internet is no small task. The triumph of .NET is that each of the capabilities mentioned (object-orientation, threading, collections, and so forth) are coordinated to facilitate just this type of development. Although Web Services are not confined to the .NET platform, the .NET Framework provides such significant productivity advantages to this cutting-edge development challenge that its success is a foregone conclusion.

Summary

Computers have no common sense. Every detail that is necessary to describe and solve a problem must be made explicit by programmers. But to reason about problems, humans need to put aside the details and concentrate on the abstract "big picture." The history of computer programming can be seen as a process of discovering better ways to organize details while keeping the abstract big picture in focus.

One route of discovery focused on data abstraction as the key to tackling large problems. Database programming languages are based on discovering the common and unique elements of data in the problem and use the transformation of input data into output data as the leading principle that will give them a bearing on a problem and its possible solution. Another line of attack focused on

behavior as the major challenge. Structured programming uses behavior as the primary structural element and emphasizes the discovery of common functions.

Object-oriented programming takes the stand that both data and behavior are equally important. Logically related data and behavior are grouped together in program elements called *types*. All instances of a given type have the same behavior, but may have different data. Integers are a type that can be added and subtracted, strings are a type that can be appended to other strings, and dogs are a type that barks at strangers. 47 and 23 are two instances of the integer type, "E pluribus unum" and "With Liberty and Justice for All" are two instances of the string type, and Lassie and Rin Tin Tin are two instances of the dog type.

The most common form of type is the *class*. An instance of a particular class is called an *object*. Object-oriented programming consists of defining the behavior of classes and creating objects and filling them with data. Naturally, this data will be instances of particular types, and the data in these instances will themselves be instances of yet other types, and so on. So an object-oriented program consists of a web of inter-related objects.This sounds confusing, but it turns out to be a very natural way to talk about problems and their solutions.

Classes can be related by a special "is-a" relationship called *inheritance*. A class that inherits from another class starts with all the characteristics of the *ancestor* class and can add data or change behavior. Since a dog is a type of mammal and all mammals have warm blood, the **Dog** class could descend from **Mammal**. The data and behavior relating to warm-bloodedness would be in the **Mammal** class, the data and behavior relating to barking at strangers in the **Dog** class. Once this was done, programmers and domain experts developing a veterinary application could talk about a problem and solution relating to body temperature by speaking of the different characteristics of **Mammal**s and **Reptile**s, rather than focusing exclusively on either a data characteristic (the blood temperature) or a behavioral characteristics (panting versus basking).

The programmer of a class can choose whether its *methods* (the functions that specify behavior) may or must be overridden by descendant classes. This aids the ability of programmers and domain experts to isolate and discuss the different abstractions in a problem. One can speak of, say, the overall procedure for an online checkout without going into the details of credit-card versus corporate-account payments. Conversely, one can implement a credit-card authorization or a corporate-account debit safe in the knowledge that they can only be accessed according to a defined interface.

The collection classes and database model of .NET make it easier to structure the web of inter-related objects that make up an OOP solution. Similarly, the

underlying framework takes care of managing memory and low-level threading issues, which are notoriously prone to disasters resulting from missed details. These facilities *do* cost some amount of performance compared to what can be done by a skilled programmer "coding to the metal," but this inherent penalty is lower than most people think. Poor performance is most often the result of inefficient design, and C# and object-orientation facilitate efficient design.

Over the years, the "typical" programming project has changed from a specialized calculation for a tolerant scientist to an information management task for a busy professional. The challenge to today's programmers is not often the efficient expression of a sophisticated mathematical model, but is more often the rapid delivery of business value to clients in a world where the definition of value is itself subject to rapid change. Perhaps the single greatest benefit of object-orientation is that it facilitates communication between programmers and clients by providing a framework in which the domain experts' natural way of speaking can lead to program structure.

3: Hello, Objects

Although it is based on C++, C# is more of a "pure" object-oriented language.

Both C++ and C# are hybrid languages, but in C# the designers felt that the hybridization was not as important as it was in C++. A hybrid language allows multiple programming styles; the reason C++ is hybrid is to support backward compatibility with the C language. Because C++ is a superset of the C language, it includes many of that language's undesirable features, which can make some aspects of C++ overly complicated.

The C# language assumes that you want to do only object-oriented programming. This means that before you can begin you must shift your mindset into an object-oriented world (unless it's already there). The benefit of this initial effort is the ability to program in a language that is simpler to learn and to use than many other OOP languages. In this chapter we'll see the basic components of a C# program and we'll learn that everything in C# is an object, even a C# program.

You manipulate objects with references

Each programming language has its own means of manipulating data. Sometimes the programmer must be constantly aware of what type of manipulation is going on. Are you manipulating the object directly, or are you dealing with some kind of indirect representation (a pointer in C or C++) that must be treated with a special syntax?

All this is simplified in C#. You treat everything as an object, so there is a single consistent syntax that you use everywhere. Although you *treat* everything as an object, the identifier you manipulate is either a variable representing the value itself or a "reference" to an object. You might imagine the latter as a television (the object) with your remote control (the reference). As long as you're holding this reference, you have a connection to the television, but when someone says "change the channel" or "lower the volume," what you're manipulating is the reference, which in turn modifies the object. If you want to move around the

room and still control the television, you take the remote/reference with you, not the television.

Also, the remote control can stand on its own, with no television. That is, just because you have a reference doesn't mean there's necessarily an object connected to it. So if you want to hold a television, you create a **Television** reference:

```
Television t;
```

But here you've created *only* the reference, not an object. If you decided to send a message to **t** at this point, you'll get an error because **t** isn't actually attached to anything (there's no television).

You must create all the objects

When you create a reference, you want to connect it with a new object. You do so, in general, with the **new** keyword. **new** says, "Make me a new one of these objects." So you can say:

```
Remote myRemote = new Remote(lastChannelWatched);
```

Not only does this mean "Make me a new **Remote**," but it also gives information about *how* to make the **Remote** by supplying some initial context.

Of course, you would have had to have programmed a **Remote** type for this code to work. In fact, that's the fundamental activity in C# programming: creating new types that represent the problem and solution to the task at hand. Learning how to do that, and gaining a familiarity with the plethora of preexisting classes in the .NET Framework Library is what you'll be learning about in the rest of this book.

Where storage lives

It's useful to visualize some aspects of how things are laid out while the program is running, in particular how memory is arranged. There are six different places to store data:

1. **Registers**. This is the fastest storage because it exists in a place different from that of other storage: inside the processor. However, the number of registers is severely limited, so registers are allocated by the JIT compiler according to its needs. You don't have direct control, nor do you see any evidence in your programs that registers even exist.

2. **The stack**. This lives in the general RAM (random-access memory) area, but has direct support from the processor via its *stack pointer*. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, second only to registers. The C# just-in-time compiler must know, while it is creating the program, the exact size and lifetime of all the data that is stored on the stack, because it must generate the code to move the stack pointer up and down. This constraint places limits on the flexibility of your programs, so while some C# storage exists on the stack—in particular, *value types* (explained shortly) and *references* to objects —C# objects themselves are not placed on the stack.

3. **The heap**. This is a general-purpose pool of memory (also in the RAM area) where all C# objects live. The nice thing about the heap is that, unlike the stack, the compiler doesn't need to know how much storage it needs to allocate from the heap or how long that storage must stay on the heap. Thus, there's a great deal of flexibility in using storage on the heap. Whenever you need to create an object, you simply write the code to create it using **new**, and the storage is allocated on the heap when that code is executed. Of course there's a price you pay for this flexibility: it takes more time to allocate heap storage than it does to allocate stack storage.

4. **Static storage**. "Static" is used here in the sense of "in a fixed location" (although it's also in RAM). Static storage contains data that is available for the entire time a program is running. You can use the **static** keyword to specify that a particular element of an object is static, but C# objects themselves are never placed in static storage.

5. **Constant storage**. Constant values are often placed directly in the program code, which is safe since they can never change. Sometimes constants are cordoned off by themselves so that they can be optionally placed in read-only memory (ROM).

6. **Non-RAM storage**. If data lives completely outside a program it can exist while the program is not running, outside the control of the program. The two primary examples of this are *serialized objects,* in which objects are turned into streams of bytes, generally to be sent to another process or machine, and *persistent objects,* in which the objects are placed on disk so they will hold their state even when the program is terminated. The trick with these types of storage is turning the objects into something that can exist on the other medium, can be resurrected

into a regular RAM-based object when necessary, and which still provides for correct behavior when a new version of the object is released. .NET Remoting provides for serialization in a number of ways and which makes huge strides towards addressing the problem of versioning. Future versions of .NET might provide even more complete solutions for persistence, such as support for database-style queries on stored objects.

Arrays in C#

Virtually all programming languages support arrays. Using arrays in C and C++ is perilous because those arrays are only blocks of memory. If a program accesses the array outside of its memory block or uses the memory before initialization (common programming errors) there will be unpredictable results.

One of the primary goals of C# is safety, so many of the problems that plague programmers in C and C++ are not repeated in C#. A C# array is guaranteed to be initialized and cannot be accessed outside of its range. The range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run-time, but the assumption is that the safety and increased productivity is worth the expense.

When you create an array of objects, you are really creating an array of references, and each of those references is automatically initialized to a special value with its own keyword: **null**. When C# sees **null**, it recognizes that the reference in question isn't pointing to an object. You must assign an object to each reference before you use it, and if you try to use a reference that's still **null,** the problem will be reported at run-time. Thus, typical array errors are prevented in C#.

You can also create an array of *value types* (which will be described next). Again, the compiler guarantees initialization because it zeroes the memory for that array.

Arrays will be covered in detail in later chapters.

Special case: value types

Unlike "pure" object-oriented languages such as Smalltalk, C# does not insist that every variable must be an object. While the performance of most systems is not determined by a single factor, the allocation of many small objects can be notoriously costly. A story goes that in the early 1990s, a manager decreed that his programming team switch to Smalltalk to gain the benefits of object-orientation; an obstinate C programmer immediately ported the application's

core matrix-manipulating algorithm to Smalltalk. The manager was pleased with this surprisingly cooperative behavior, as the programmer made sure that everyone knew that he was integrating the new Smalltalk code that very afternoon and running it through the stress test before making it the first Smalltalk code to be integrated into the production code. Twenty-four hours later, when the matrix manipulation had not completed, the manager realized that he'd been had, and never spoke of Smalltalk again.

When Java became popular, many people predicted similar performance problems. However, Java has "primitive" types for integers and characters and so forth and many people have found that this has been sufficient to make Java appropriate for almost all performance-oriented tasks. C# goes a step beyond; not only are values (rather than classes) used for basic numeric types, developers can create new value types in the form of enumerations (**enum**s) and structures (**struct**s).

Value types can be transparently converted to and from object references via a process known as "boxing." This is a nice advantage of C# over Java, where turning a primitive type into an object reference requires an explicit method call. Boxing is described in more detail on Page 65.

You never need to destroy an object

In most programming languages, the concept of the lifetime of a variable occupies a significant portion of the programming effort. How long does the variable last? If you are supposed to destroy it, when should you? Confusion over variable lifetimes can lead to a lot of bugs, and this section shows how C# greatly simplifies the issue by doing all the cleanup work for you.

Scoping

Most procedural languages have the concept of *scope*. This determines both the visibility and lifetime of the names defined within that scope. In C, C++, and C#, scope is determined by the placement of curly braces **{}**. So for example:

```
{
  int x = 12;
  /* only x available */
  {
    int q = 96;
    /* both x & q available */
```

```
  }
  /* only x available */
  /* q "out of scope" */
}
```

A variable defined within a scope is available only to the end of that scope.

Indentation makes C# code easier to read. Since C# is a free-form language, the extra spaces, tabs, and carriage returns do not affect the resulting program.

Note that you *cannot* do the following, even though it is legal in C and C++:

```
{
  int x = 12;
  {
    int x = 96; /* illegal */
  }
}
```

The compiler will announce that the variable **x** has already been defined. Thus the C and C++ ability to "hide" a variable in a larger scope is not allowed because the C# designers thought that it led to confusing programs.

Scope of objects

C# objects do not have the same lifetimes as value types such as structs. When you create a C# object using **new**, it hangs around past the end of the scope. Thus if you use:

```
{
  Television t = new Television();
} /* end of scope */
```

the reference **t** vanishes at the end of the scope. However, the **Television** object that **t** was pointing to is still occupying memory. In this bit of code, there is no way to access the object because the only reference to it is out of scope. In later chapters you'll see how the reference to the object can be passed around and duplicated during the course of a program.

It turns out that because objects created with **new** stay around for as long as you want them, a whole slew of C++ programming problems simply vanish in C#. The hardest problems seem to occur in C++ because you don't get any help from the language in making sure that the objects are available when they're needed. And more important, in C++ you must make sure that you destroy the objects when you're done with them.

That brings up an interesting question. If C# leaves the objects lying around, what keeps them from filling up memory and halting your program? This is exactly the kind of problem that would occur in C++. Here is where a bit of magic happens. The .NET runtime has a *garbage collector*, which looks at all the objects that were created with **new** and figures out which ones are not being referenced anymore. Then it releases the memory for those objects, so the memory can be used for new objects. This means that you never need to worry about reclaiming memory yourself. You simply create objects, and when you no longer need them they will go away by themselves. This eliminates a certain class of programming problem: the so-called "memory leak," in which a programmer forgets to release memory.

Creating new data types: class

If everything is an object, what determines how a particular class of object looks and behaves? Put another way, what establishes the *type* of an object? You might expect there to be a keyword called "type," and that certainly would have made sense. Historically, however, most object-oriented languages have used the keyword **class** to mean "I'm about to tell you what a new type of object looks like." The **class** keyword (which is so common that it will not be emboldened throughout this book) is followed by the name of the new type. For example:

```
class ATypeName { /* class body goes here */ }
```

This introduces a new type, so you can now create an object of this type using **new**:

```
ATypeName a = new ATypeName();
```

In **ATypeName**, the class body consists only of a comment (the stars and slashes and what is inside, which will be discussed later in this chapter), so there is not too much that you can do with it. In fact, you cannot tell it to do much of anything (that is, you cannot send it any interesting messages) until you define some methods for it.

Fields, properties, and methods

When you define a class, you can put three types of elements in your class: data members (sometimes called *fields*), member functions (typically called *methods*), and properties. A data member is an object of any type that you can communicate with via its reference. It can also be a value type (which isn't a reference). If it is a reference to an object, you must initialize that reference to connect it to an actual

object (using **new**, as seen earlier) in a special function called a *constructor* (described fully in Chapter 4). If it is a primitive type you can initialize it directly at the point of definition in the class. (As you'll see later, references can also be initialized at the point of definition.)

Each object keeps its own storage for its data members; the data members are not shared among objects. Here is an example of a class with some data members:

```
public class DataOnly {
  public int i;
  public float f;
  public bool b;
  private string s;
}
```

This class doesn't *do* anything, but you can create an object:

```
DataOnly d = new DataOnly();
```

Both the classname and the fields except **s** are preceded by the word **public**. This means that they are visible to all other objects. You can assign values to data members that are visible, but you must first know how to refer to a member of an object. This is accomplished by stating the name of the object reference, followed by a period (dot), followed by the name of the member inside the object:

```
objectReference.member
```

For example:

```
d.i = 47;
d.f = 1.1;
d.b = false;
```

However, the **string s** field is marked **private** and is therefore not visible to any other object (later, we'll discuss other *access modifiers* that are intermediate between **public** and **private**). If you tried to write:

```
d.s = "asdf";
```

you would get a compile error. Data hiding seems inconvenient at first, but is *so* helpful in a program of any size that the default visibility of fields is **private**.

It is also possible that your object might contain other objects that contain data you'd like to modify. For this, you just keep "connecting the dots." For example:

```
myPlane.leftTank.capacity = 100;
```

The **DataOnly** class cannot do much of anything except hold data, because it has no member functions (methods). To understand how those work, you must first understand *arguments* and *return values*, which will be described shortly.

Default values for value types

When a value type is a member of a class, it is guaranteed to get a default value if you do not initialize it:

Value type	Size in bits	Default
bool	**4**	**false**
char	**8**	**'\u0000' (null)**
byte, sbyte	**8**	**(byte)0**
short, ushort	**8**	**(short)0**
int, uint	**32**	**0**
long, ulong	**64**	**0L**
float	**8**	**0.0f**
double	**64**	**0.0d**
decimal	**96**	**0**
string	**160 minimum**	**'' (empty)**
object	**64 minimum overhead**	**null**

Note carefully that the default values are what C# guarantees when the variable is used *as a member of a class*. This ensures that member variables of primitive types will always be initialized (something C++ doesn't do), reducing a source of bugs. However, this initial value may not be correct or even legal for the program you are writing. It's best to always explicitly initialize your variables.

This guarantee doesn't apply to "local" variables—those that are not fields of a class. Thus, if within a function definition you have:

```
int x;
```

you must have an appropriate value to **x** before you use it. If you forget, C# definitely improves on C++: you get a compile-time error telling you the variable might not have been initialized. (Many C++ compilers will warn you about uninitialized variables, but in C# these are errors.)

The previous table contains some rows with multiple entries, e.g., **short** and **ushort**. These are signed and unsigned versions of the type. An unsigned version

of an integral type can take any value between 0 and $2^{bitsize-1}$ while a signed version can take any value between $-2^{bitsize-1}$ to $2^{bitsize-1}-1$.

Methods, arguments, and return values

Up until now, the term *function* has been used to describe a named subroutine. The term that is more commonly used in C# is *method,* as in "a way to do something." If you want, you can continue thinking in terms of functions. It's really only a syntactic difference, but from now on "method" will be used in this book rather than "function."

Methods in C# determine the messages an object can receive. In this section you will learn how simple it is to define a method.

The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

```
returnType MethodName( /* Argument list */ ) {
  /* Method body */
}
```

The return type is the type of the value that pops out of the method after you call it. The argument list gives the types and names for the information you want to pass into the method. The method name and argument list together uniquely identify the method.

Methods in C# can be created only as part of a class. A method can be called only for an object,[1] and that object must be able to perform that method call. If you try to call the wrong method for an object, you'll get an error message at compile time. You call a method for an object by naming the object followed by a period (dot), followed by the name of the method and its argument list, like this: **objectName.MethodName(arg1, arg2, arg3)**. For example, suppose you have a method **F( )** that takes no arguments and returns a value of type **int**. Then, if you have an object called **a** for which **F( )** can be called, you can say this:

```
int x = a.F();
```

The type of the return value must be compatible with the type of **x**.

---

[1] **static** methods, which you'll learn about soon, can be called *for the class*, without an object.

This act of calling a method is commonly referred to as *sending a message to an object*. In the above example, the message is **F( )** and the object is **a**. Object-oriented programming is often summarized as simply "sending messages to objects."

The argument list

The method argument list specifies what information you pass into the method. As you might guess, this information—like everything else in C#—takes the form of objects. So, what you must specify in the argument list are the types of the objects to pass in and the name to use for each one. As in any situation in C# where you seem to be handing objects around, you are actually passing references. The type of the reference must be correct, however. If the argument is supposed to be a **string**, what you pass in must be a string.

Consider a method that takes a **string** as its argument. Here is the definition, which must be placed within a class definition for it to be compiled:

```
int Storage(string s) {
  return s.Length * 2;
}
```

This method tells you how many bytes are required to hold the information in a particular **string.** (Each **char** in a **string** is 16 bits, or two bytes, long, to support Unicode characters[2].)The argument is of type **string** and is called **s**. Once **s** is passed into the method, you can treat it just like any other object. (You can send messages to it.) Here, the **Length** property is used, which is one of the properties of **string**s; it returns the number of characters in a string.

You can also see the use of the **return** keyword, which does two things. First, it means "leave the method, I'm done." Second, if the method produces a value, that value is placed right after the **return** statement. In this case, the return value is produced by evaluating the expression **s.Length * 2**.

You can return any type you want, but if you don't want to return anything at all, you do so by indicating that the method returns **void**. Here are some examples:

```
boolean Flag() { return true; }
```

---

[2] The bit-size and interpretation of **char**s can actually be manipulated by a class called **Encoding** and this statement refers to the default "Unicode Transformation Format, 16-bit encoding form" or UTF-16. Other encodings are UTF-8 and ASCII, which use 8 bits to define a character.

```
float NaturalLogBase() { return 2.718f; }
void Nothing() { return; }
void Nothing2() {}
```

When the return type is **void**, then the **return** keyword is used only to exit the method, and is therefore unnecessary when you reach the end of the method. You can return from a method at any point, but if you've given a non-**void** return type then the compiler will force you (with error messages) to return the appropriate type of value regardless of where you return.

At this point, it can look like a program is just a bunch of objects with methods that take other objects as arguments and send messages to those other objects. That is indeed much of what goes on, but in the following chapter you'll learn how to do the detailed low-level work by making decisions within a method. For this chapter, sending messages will suffice.

Attributes and meta-behavior

The most intriguing low-level feature of the .NET Runtime is the attribute, which allows you to specify arbitrary meta-information to be associated with code elements such as classes, types, and methods. Attributes are specified in C# using square brackets just before the code element. Adding an attribute to a code element doesn't change the behavior of the code element; rather, programs can be written which say "For all the code elements that have this attribute, do this behavior." The most immediately powerful demonstration of this is the **[WebMethod]** attribute which within Visual Studio .NET is all that is necessary to trigger the exposure of that method as a Web Service.

Attributes can be used to simply tag a code element, as with **[WebMethod]**, or they can contain parameters that contain additional information. For instance, this example shows an **XMLElement** attribute that specifies that, when serialized to an XML document, the **FlightSegment[ ]** array should be created as a series of individual **FlightSegment** elements:

```
[XmlElement(
   ElementName = "FlightSegment")]
   public FlightSegment[] flights;
```

Attributes will be explained in Chapter 13 and XML serialization will be covered in Chapter 17.

Delegates

In addition to classes and value types, C# has an object-oriented type that specifies a method *signature*. A method's signature consists of its argument list and its return type. A **delegate** is a type that allows any method whose signature is identical to that specified in the delegate definition to be used as an "instance" of that delegate. In this way, a method can be used as if it were a variable – instantiated, assigned to, passed around in reference form, etc. C++ programmers will naturally think of delegates as being quite analogous to function pointers.

In this example, a delegate named **BluffingStrategy** is defined:

```
delegate void BluffingStrategy(PokerHand x);

public class BlackBart{
   public void SnarlAngrily(PokerHand y){ … }
   public int AnotherMethod(PokerHand z){ … }
}
public class SweetPete{
   public void YetAnother(){ … }
   public static void SmilePleasantly(PokerHand z){ … }
}
```

The method **BlackBart.SnarlAngrily( )** could be used to instantiate the **BluffingStrategy** delegate, as could the method **SweetPete.SmilePleasantly( )**. Both of these methods do not return anything (they return **void**) and take a **PokerHand** as their one-and-only parameter—the exact method signature specified by the **BluffingStrategy** delegate.

Neither **BlackBart**.**AnotherMethod( )** nor **SweetPete.YetAnother( )** can be used as **BluffingStrategy**s, as these methods have different signatures than **BluffingStrategy**. **BlackBart.AnotherMethod( )** returns an **int** and **SweetPete.YetAnother( )** does not take a **PokerHand** argument.

Instantiating a reference to a delegate is just like making a reference to a class:

```
BluffingStrategy bs =
  new BluffingStrategy(SweetPete.SmilePleasantly);
```

The left-hand size contains a declaration of a variable **bs** of type delegate **BluffingStrategy**. The right-hand side *specifies* a method; it does not actually *call* the method **SweetPete.SmilePleasantly( )**.

To actually call the delegate, you put parentheses (with parameters, if appropriate) after the variable:

```
bs(); //equivalent to: SweetPete.SmilePleasantly()
```

Delegates are a major element in programming Windows Forms, but they represent a major design feature in C# and are useful in many situations.

Properties

Fields should, essentially, never be available directly to the outside world. Mistakes are often made when a field is assigned to; the field is supposed to store a distance in metric not English units, strings are supposed to be all lowercase, etc. However, such mistakes are often not *found* until the field is used at a much later time (like, say, when preparing to enter Mars orbit). While such logical mistakes cannot be discovered by any automatic means, discovering them can be made easier by only allowing fields to be accessed via methods (which, in turn, can provide additional sanity checks and logging traces).

C# allows you to give your classes the appearance of having fields directly exposed but in fact hiding them behind method invocations. These **Property** fields come in two varieties: read-only fields that cannot be assigned to, and the more common read-and-write fields. Additionally, properties allow you to use a different type internally to store the data from the type you expose. For instance, you might wish to expose a field as an easy-to-use **bool**, but store it internally within an efficient **BitArray** class (discussed in Chapter 9).

Properties are specified by declaring the type and name of the Property, followed by a bracketed code block that defines a **get** code block (for retrieving the value) and a **set** code block. Read-only properties define only a **get** code block (it is legal, but not obviously useful, to create a write-only property by defining just set). The **get** code block acts as if it were a method defined as taking no arguments and returning the type defined in the Property declaration; the **set** code block acts as if it were a method returning void that takes an argument named **value** of the specified type. Here's an example of a read-write property called **PropertyName** of type **MyType**.

```
//MyClass.cs
//Demonstrates a property
class MyClass {
   MyType myInternalReference;

   //Begin property definition
```

```
   public MyType PropertyName{
      get {
         //logic
         return myInternalReference;
      }

      set{
         //logic
         myInternalReference = value;
      }
   }
   //End of property definition
}//(Not intended to compile – MyType does not exist)
```

To use a Property, you access the name of the property directly:

```
myClassInstance.MyProperty = someValue; //Calls "set"
MyType t = myClassInstance.MyProperty;  //Calls "get"
```

One of the most common rhetorical questions asked by Java advocates is "What's the point of properties when all you have to do is have a naming convention such as Java's **getPropertyName( )** and **setPropertyName( )**? It's needless complexity." The C# compiler in fact *does* create just such methods in order to implement properties (the methods are called **get_PropertyName( )** and **set_PropertyName( )**). This is a theme of C# — direct language support for features that are implemented, not directly in Microsoft Intermediate Language (MSIL – the "machine code" of the .NET runtime), but via code generation. Such "syntactic sugar" could be removed from the C# language without actually changing the set of problems that can be solved by the language; they "just" make certain tasks easier. Properties make the code a little easier to read and make reflection-based meta-programming (discussed in Chapter 13) a little easier. Not every language is designed with ease-of-use as a major design goal and some language designers feel that syntactic sugar ends up confusing programmers. For a major language intended to be used by the broadest possible audience, C#'s language design is appropriate; if you want something boiled down to pure functionality, there's talk of LISP being ported to .NET.

Creating new value types

In addition to creating new classes, you can create new value types. One nice feature that C# enjoys is the ability to automatically *box* value types. Boxing is the process by which a value type is transformed into a reference type and vice versa. Value types can be automatically transformed into references by boxing and a

boxed reference can be transformed back into a value, but reference types cannot be automatically transformed into value types.

Enumerations

An enumeration is a set of related values: Up-Down, North-South-East-West, Penny-Nickel-Dime-Quarter, etc. An enumeration is defined using the **enum** keyword and a code block in which the various values are defined. Here's a simple example:

```
enum UpOrDown{ Up, Down }
```

Once defined, an enumeration value can be used by specifying the enumeration type, a dot, and then the specific name desired:

```
UpOrDown coinFlip = UpOrDown.Up;
```

The names within an enumeration are actually numeric values. By default, they are integers, whose value begins at zero. You can modify both the type of storage used for these values and the values associated with a particular name. Here's an example, where a **short** is used to hold different coin values:

```
enum Coin: short{
  Penny = 1, Nickel = 5, Dime = 10, Quarter = 25
}
```

Then, the names can be cast to their implementing value type:

```
short change = (short) (Coin.Penny + Coin.Quarter);
```

This will result in the value of **change** being 26.

It is also possible to do bitwise operations on enumerations that are given compatible:

```
enum Flavor{
  Vanilla = 1, Chocolate = 2, Strawberry = 4, Coffee = 8
}
...etc...
Flavor conePref = Flavor.Vanilla | Flavor.Coffee;
```

Structs

A **struct** (short for "structure") is very similar to a class in that it can contain fields, properties, and methods. However, **struct**s are value types and are created on the stack (see page 50); you cannot inherit from a **struct** or have your **struct**

inherit from any class (although a **struct** can implement an interface), and **struct**s have limited constructor and destructor semantics.

Typically, **struct**s are used to aggregate a relatively small amount of logically related fields. For instance, the Framework SDK contains a **Point** structure that has **X** and **Y** properties. Structures are declared in the same way as classes. This example shows what might be the start of a **struct** for imaginary numbers:

```
struct ImaginaryNumber{
  double real;
  public double Real{
    get { return real; }
    set { real = value; }
  }

  double i;
  public double I{
    get { return i; }
    set { i = value; }
  }
}
```

Boxing and Unboxing

The existence of both reference types (classes) and value types (structs, enums, and primitive types) is one of those things that object-oriented academics love to sniff about, saying that the distinction is too much for the poor minds that are entering the field of computer programming. Nonsense. As discussed previously, the key distinction between the two types is where they are stored in memory: value types are created on the stack while classes are created on the heap and are referred to by one or more stack-based references (see Page 50).

To revisit the metaphor from that section, a class is like a television (the object created on the heap) that can have one or more remote controls (the stack-based references), while a value-type is like a thought: when you give it to someone, you are giving them a copy, not the original. This difference has two major consequences: *aliasing* (which will be visited in depth in Chapter 4) and the lack of an object reference. As was discussed on Page 49, you manipulate objects with a reference: since value types do not *have* such a reference, you must somehow create one before doing anything with a value type that is more sophisticated than basic math. One of C#'s notable advantages over Java is that C# makes this process transparent.

The processes called *boxing* and *unboxing* wrap and unwrap a value type in an object. Thus, the **int** primitive type can be *boxed* into an object of the class **Int32**, a **bool** is boxed into a **Boolean**, etc. Boxing and unboxing happen transparently between a variable declared as the value type and its equivalent class type. Thus, you can write code like the following:

```
bool valueType1 = true;
Boolean referenceType1 = b;       //Boxing
bool valueType2 = referenceType1; //Unboxing
```

The utility of boxing and unboxing will become more apparent in Chapter 10's discussion of collection classes and data structures, but there is one value type for which the benefits of boxing and unboxing become apparent immediately: the **string**.

Strings and formatting

Strings are probably the most manipulated type of data in computer programs. Sure, numbers are added and subtracted, but strings are unusual in that their structure is of so much interest: we search for substrings, change the case of letters, construct new strings from old strings, and so forth. Since there are so many operations that one wishes to do on strings, it is obvious that they must be implemented as classes. Strings are incredibly common and are often at the heart of the innermost loops of programs, so they must be as efficient as possible, so it is equally obvious that they must be implemented as stack-based value types. Boxing and unboxing allow these conflicting requirements to coexist: **string**s are value types, while the **String** class provides a plethora of powerful methods.

The single-most used method in the **String** class must be the **Format** method, which allows you to specify that certain patterns in a string be replaced by other string variables, in a certain order, and formatted in a certain way. For instance, in this snippet:

```
string w = "world";
string s = String.Format("Hello, {0}", w);
```

The value of **s** would be "Hello, world", as the value of the variable **w** is substituted for the pattern `{0}`. Such substitutions can be strung out indefinitely:

```
string h = "hello";
string w = "world";
string hw = "how";
string r = "are";
```

```
string u = "you";
string q = "?";
string s = String.Format("{0} {1}, {2} {3} {4}{5}"
             , h, w, hw, r, u, q);
```

gives **s** the value of "hello world, how are you?". This variable substitution pattern will be used often in this book, particularly in the **Console.WriteLine( )** method that is used to write strings to the console.

Additionally, .NET provides for powerful formatting of numbers, dates, and times. This formatting is locale-specific, so on a computer set to use United States conventions, currency would be formatted with a '$' character, while on a machine configured for Europe, the '€' would be used (as powerful a library as it is, it only formats the string, it cannot do the actual conversion calculation between dollars and euros!). A complete breakdown of the string formatting patterns is beyond the scope of this book, but in addition to the simple variable substitution pattern shown above, there are two number-formatting patterns that are very helpful:

```
double doubleValue = 123.456;
Double doubleObject = doubleValue; //Boxed
string s = doubleObject.ToString("####.#"); //Unboxed
string s2 = doubleObject.ToString("0000.0"); //Unboxed
```

Again, this example relies on boxing and unboxing to transparently convert, first, the **doubleValue** value type into the **doubleObject** object of the **Double** class. Then, the **ToString( )** method, which supports string formatting patterns, creates two **String** objects which are unboxed into **string** value types. The value of **s** is "123.5" and the value of **s2** is "0123.5". In both cases, the digits of the boxed **Double** object (that has the value 123.456) are substituted for the '#' and '0' characters in the formatting pattern. The '#' pattern does not output the non-significant 0 in the thousands place, while the '0' pattern does. Both patterns, with only one character after the decimal point, output a rounded value for the number.

Building a C# program

There are several other issues you must understand before seeing your first C# program.

Name visibility

A problem in any programming language is the control of names. If you use a name in one module of the program, and another programmer uses the same

name in another module, how do you distinguish one name from another and prevent the two names from "clashing?" In C this is a particular problem because a program is often an unmanageable sea of names. C++ classes (on which C# classes are based) nest functions within classes so they cannot clash with function names nested within other classes. However, C++ still allowed global data and global functions, and the class names themselves could conflict, so clashing was still possible. To solve this problem, C++ introduced *namespaces* using additional keywords.

In C#, the **namespace** keyword is followed by a code block (that is, a pair of curly brackets with some amount of code within them). Unlike Java, there is no relationship between the namespace and class names and directory and file structure. Organizationally, it often makes sense to gather all the files associated with a single namespace into a single directory and to have a one-to-one relationship between class names and files, but this is strictly a matter of preference. Throughout this book, our example code will often combine multiple classes in a single compilation unit (that is, a single file) and we will typically not use namespaces, but in professional development, you should avoid such space-saving choices.

Namespaces can, and should, be nested. By convention, the outermost namespace is the name of your organization, the next the name of the project or system as a whole, and the innermost the name of the specific grouping of interest. Here's an example:

```
namespace ThinkingIn {
   namespace CSharp {
      namespace Chap2 {
         //class and other type declarations go here
      }
   }
}
```

Since namespaces are publicly viewable, they should start with a capital letter and then use "camelcase" capitalization (for instance, **ThinkingIn**).

Namespaces are navigated using dot syntax: **ThinkingIn.CSharp.Chap2** may even be declared in this manner:

```
namespace ThinkingIn.CSharp.Chap2{ … }
```

Using other components

Whenever you want to use a predefined class in your program, the compiler must know how to locate it. The first place the compiler looks is the current program file, or *assembly*. If the assembly was compiled from multiple source code files, and the class you want to use was defined in one of them, you simply use the class.

What about a class that exists in some other assembly? You might think that there ought to just be a place where all the assemblies that are used by all the programs on the computer are stored and the compiler can look in that place when it needs to find a class. But this leads to two problems. The first has to do with names; imagine that you want to use a class of a particular name, but more than one assembly uses that name (for instance, probably a lot of programs define a class called **User**). Or worse, imagine that you're writing a program, and as you're building it you add a new class to your library that conflicts with the name of an existing class.

To solve this problem, you must eliminate all potential ambiguities. This is accomplished by telling the C# compiler exactly what classes you want using the **using** keyword. **using** tells the compiler to recognize the names in a particular *namespace*, which is just a higher-level organization of names. The .NET Framework SDK has more than 100 namespaces, such as **System.Xml** and **System.Windows.Forms** and **Microsoft.Csharp**. By adhering to some simple naming conventions, it is highly unlikely that name clashes will occur and, if they do, there are simple ways to remove the ambiguity between namespaces.

Java and C++ programmers should understand that namespaces and **using** are different than **import** or **#include**. Namespaces and **using** are strictly about naming concerns at compile-time, while Java's **import** statement relates also to finding the classes at run-time, while C++'s **#include** moves the referenced text into the local file.

The second problem with relying on classes stored in a different assembly is the threat that the user might inadvertently replace the version your class needs with another version of the assembly with the same name but with different behavior. This was the root cause of the Windows problem known as "DLL Hell." Installing or updating one program would change the version of some widely-used shared library.

To solve this problem, when you compile an assembly that depends on another, you can embed into the dependent assembly a reference to the *strong name* of the other assembly. This name is created using public-key cryptography and,

along with infrastructure support for a *Global Assembly Cache* that allows for assemblies to have the same name but different versions, gives .NET an excellent basis for overcoming versioning and trust problems. An example of strong naming and the use of the GAC begins on Page 532.

The static keyword

Ordinarily, when you create a class you are describing how objects of that class look and how they will behave. You don't actually get anything until you create an object of that class with **new**, and at that point data storage is created and methods become available.

But there are two situations in which this approach is not sufficient. One is if you want to have only one piece of storage for a particular piece of data, regardless of how many objects are created, or even if no objects are created. The other is if you need a method that isn't associated with any particular object of this class. That is, you need a method that you can call even if no objects are created. You can achieve both of these effects with the **static** keyword. When you say something is **static**, it means that data or method is not tied to any particular object instance of that class. So even if you've never created an object of that class you can call a **static** method or access a piece of **static** data. With ordinary, non-**static** data and methods you must create an object and use that object to access the data or method, since non-**static** data and methods must know the particular object they are working with. Of course, since **static** methods don't need any objects to be created before they are used, they cannot *directly* access non-**static** members or methods by simply calling those other members without referring to a named object (since non-**static** members and methods must be tied to a particular object).

Some object-oriented languages use the terms *class data* and *class methods*, meaning that the data and methods exist for any and all objects of the class.

To make a data member or method **static**, you simply place the keyword before the definition. For example, the following produces a **static** data member and initializes it:

```
class StaticTest {
    public static int i = 47;
}
```

Now even if you make two **StaticTest** objects, there will still be only one piece of storage for **StaticTest.i.** Both objects will share the same **i**. Consider:

```
StaticTest st1 = new StaticTest();
```

```
StaticTest st2 = new StaticTest();
```

At this point, both **st1** and **st2** have access to the same '47' value of **StaticTest.i** since they refer to the same piece of memory.

To reference a static variable, you use the dot-syntax, but instead of having an object reference on the left side, you use the class name.

```
StaticTest.i++;
```

The **++** operator increments the variable. At this point, both **st1** and **st2** would see **StaticTest.i** as having the value 48.

Similar logic applies to static methods. You refer to a static method using **ClassName.Method( )**. You define a static method in a similar way:

```
class StaticFun {
  public static void Incr() { StaticTest.i++; }
}
```

You can see that the **StaticFun** method **Incr( )** increments the **static** data **i**.

While **static**, when applied to a data member, definitely changes the way the data is created (one for each class vs. the non-**static** one for each object), when applied to a method it's not so dramatic. An important use of **static** for methods is to allow you to call that method without creating an object. This is essential, as we will see, in defining the **Main( )** method that is the entry point for running an application.

Like any method, a **static** method can create or use named objects of its type, so a **static** method is often used as a "shepherd" for a flock of instances of its own type.

Putting it all together

Let's write a program. It starts by printing a string, and then the date, using the **DateTime** class from the .NET Framework SDK. Note that an additional style of comment is introduced here: the '//', which is a comment until the end of the line:

```
//:c03:HelloDate.cs
using System;

namespace ThinkingIn.CSharp.Chap03{
  public class HelloDate {
```

```
    public static void Main() {
      Console.WriteLine("Hello, it's: ");
      Console.WriteLine(DateTime.Now);
    }
  }
}///:~
```

At the beginning of each program file, you place **using** statements to bring in the namespaces of any classes you'll need for the code in that file.

If you are working with the downloaded .NET Framework SDK, there is a Microsoft Help file that can be accessed with **ms-help://ms.netframeworksdk**, if using Visual Studio .NET, there is an integrated help system. If you navigate to **ms-help://MS.NETFrameworkSDK/cpref/html/frlrfSystem.htm**, you'll see the contents of the **System** namespace. One of them is the **Console** class. If you open this subject and then click on **Console.Members**, you'll see a list of public properties and methods. In the case of the **Console** class, all of them are marked with an "S" indicating that they are static.

One of the static methods of **Console** is **WriteLine( )**. Since it's a static method, you don't need to create an object to use it. Thus, if you've specified **using System;** you can write **Console.WriteLine("Something")** whenever you want to print something to the console. Alternately, in *any* C# program, you can specify the *fully qualified* name **System.Console.WriteLine("Something")** even if you have not written **using System**.

Every program must have what's called an *entry point*, a method which starts up things. In C#, the entry point is always a static method called **Main( )**. **Main( )** can be written in several different ways:

```
static void Main(){ … }
static void Main(string[] args){ … }
static int Main(){ … }
static int Main(string[] args){ … }
```

If you wish to pass in parameters from the command-line to your program, you should use one of the forms that takes an array of command-line arguments. **args[0]** will be the first argument *after* the name of the executable.

Traditionally, programs return zero if they ran successfully and some other integer as an error code if they failed. C#'s exceptions are infinitely superior for communicating such problems, but if you are writing a program that you wish to

program with batch files (which pay attention to the return value of a program), you may wish to use the version of **Main( )** that returns an integer.

The line that prints the date illustrates the behind-the-scenes complexity of even a simple object-oriented call:

```
Console.WriteLine(DateTime.Now);
```

Consider the argument: if you browse the documentation to the **DateTime** structure, you'll discover that it has a static property **Now** of type **DateTime**. As this property is read, the .NET Runtime reads the system clock, creates a new **DateTime** value to store the time, and returns it. As soon as that property **get** finishes, the **DateTime** struct is passed to the static method **WriteLine( )** of the **Console** class. If you use the helpfile to go to that method's definition, you'll see many different *overloaded* versions of **WriteLine( )**, one which takes a **bool**, one which takes a **char**, etc. You won't find one that takes a **DateTime**, though.

Since there is no overloaded version that takes the exact type of the **DateTime** argument, the runtime looks for *ancestors* of the argument type. All **struct**s are defined as descending from type **ValueType**, which in turn descends from type **object**. There is not a version of **WriteLine( )** that takes a **ValueType** for an argument, but there *is* one that takes an **object**. It is this method that is called, passing in the **DateTime** structure.

Back in the documentation for **WriteLine( )**, it says it calls the **ToString( )** method of the **object** passed in as its argument. If you browse to **Object.ToString( )**, though, you'll see that the default representation is just the fully qualified name of the object. But when run, this program doesn't print out "System.DateTime," it prints out the time value itself. This is because the implementers of the **DateTime** class *overrode* the default implementation of **ToString( )** and the call within **WriteLine( )** is resolved *polymorphically* by the **DateTime** implementation, which returns a culture-specific **string** representation of its value to be printed to the **Console**.

If some of that doesn't make sense, don't worry – almost every aspect of object-orientation is at work within this seemingly trivial example.

Compiling and running

To compile and run this program, and all the other programs in this book, you must first have a command-line C# compiler. We *strongly* urge you to refrain from using Microsoft Visual Studio .NET's GUI-activated compiler for compiling the sample programs in this book. The less that is between raw text code and the

running program, the more clear the learning experience. Visual Studio .NET introduces additional files to structure and manage projects, but these are not necessary for the small sample programs used in this book. Visual Studio .NET has some great tools that ease certain tasks, like connecting to databases and developing Windows Forms, but these tools should be used to relieve drudgery, not as a substitute for knowledge. The one big exception to this is the "IntelliSense" feature of the Visual Studio .NET editor, which pops up information on objects and parameters faster than you could possibly search through the .NET documentation.

A command-line C# compiler is included in Microsoft's .NET Framework SDK, which is available for free download at msdn.microsoft.com/downloads/ in the "Software Development Kits" section. A command-line compiler is also included within Microsoft Visual Studio .NET. The command-line compiler is **csc.exe**. Once you've installed the SDK, you should be able to run **csc** from a command-line prompt.

In addition to the command-line compiler, you should have a decent text editor. Some people seem satisfied with Windows Notepad, but most programmers prefer either the text editor within Visual Studio.NET (just use File/Open... and Save... to work directly with text files) or a third-party programmer's editor. All the code samples in this book were written with Visual SlickEdit from MicroEdge (another favorite is Computer Solution Inc.'s $35 shareware UltraEdit).

Once the Framework SDK is installed, download and unpack the source code for this book (you can find it at www.ThinkingIn.net). This will create a subdirectory for each chapter in the book. Move to the subdirectory **c03** and type:

```
csc HelloDate.cs
```

You should see a message that specifies the versions of the C# compiler and .NET Framework that are being used (the book was finished with C# Compiler version 7.10.2215.1 and .NET Framework version 1.1.4322). There should be no warnings or errors; if there are, it's an indication that something went wrong with the SDK installation and you need to investigate those problems.

On the other hand, if you just get your command prompt back, you can type:

```
HelloDate
```

and you'll get the message and the date as output.

This is the process you can use to compile and run each of the programs in this book. A source file, sometimes called a *compilation unit*, is compiled by **csc** into

a .NET *assembly*. If the compilation unit has a **Main( )**, the assembly will default to have an extension of **.exe** and can be run from the command-line just as any other program.

Fine-tuning compilation

An assembly may be generated from more than one compilation unit. This is done by simply putting the names of the additional compilation units on the command-line (**csc FirstClass.cs SecondClass.cs** etc.). You can modify the name of the assembly with the **/out:** argument. If more than one class has a **Main( )** defined, you can specify which one is intended to be the entry point of the assembly with the **/main:** argument.

Not every assembly needs to be a stand-alone executable. Such assemblies should be given the **/target:library** argument and will be compiled into an assembly with a **.DLL** extension.

By default, assemblies "know of" the standard library reference **mscorlib.dll**, which contains the majority of the .NET Framework SDK classes. If a program uses a class in a namespace *not* within the **mscorlib.dll** assembly, the **/reference:** argument should be used to point to the assembly.

The Common Language Runtime

You do not need to know this. But we bet you're curious.

The .NET Framework has several layers of abstraction, from very high-level libraries such as Windows Forms and the SOAP Web Services support, to the core libraries of the SDK:

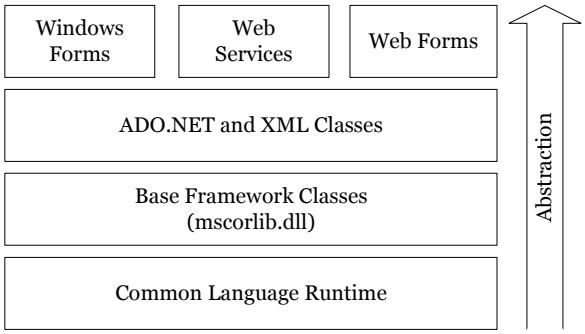


Figure 3-1: The layered architecture of the .NET Framework

Everything in this diagram except the Common Language Runtime (CLR) is stored on the computer in Common Intermediate Language (CIL, sometimes

referred to as Microsoft Intermediate Language, or MSIL, or sometimes just as IL), a very simple "machine code" for an abstract computer.

The C# compiler, like all .NET language compilers, transforms human-readable source code into CIL, not the actual opcodes of any particular CPU. An assembly consists of CIL, metadata describing the assembly, and optional resources. We'll discuss metadata in detail in Chapter 13 while resources will be discussed in Chapter 14.

The role of the Common Language Runtime can be boiled down to "mediate between the world of CIL and the world of the actual platform." This requires several components:

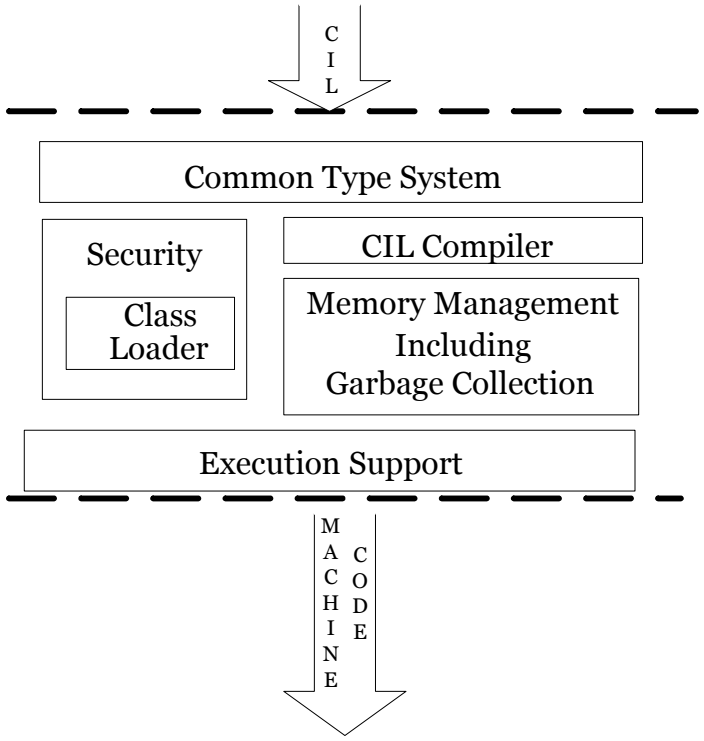


Figure 3-2: "Below" the level of CIL, all .NET languages are similar

Different CPUs and languages have traditionally represented strings in different ways and numeric types using values of different bit-lengths. The value proposition of .NET is "Any language, one platform" (in contrast with Java's value proposition of "Any platform, one language.") In order to assure that all languages can interoperate seamlessly .NET provides a uniform definition of

several basic types in the Common Type System. Once "below" this level, the human-readable language in which a module was originally written is irrelevant.

The next three listings show the transformation of a simple method from C# to CIL to Pentium machine code.

```
class Simple{
   public static void Main(){
      int sum = 0;
      for(int i = 0; i < 5; i++){
         sum += i;
      }
      Console.WriteLine(sum);
   }
}
```

becomes in CIL:

```
.method public hidebysig static void  Main() cil managed{
  .entrypoint
  // Code size       25 (0x19)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.0
  IL_0003:  stloc.1
  IL_0004:  br.s       IL_000e
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  add
  IL_0009:  stloc.0
  IL_000a:  ldloc.1
  IL_000b:  ldc.i4.1
  IL_000c:  add
  IL_000d:  stloc.1
  IL_000e:  ldloc.1
  IL_000f:  ldc.i4.5
  IL_0010:  blt.s      IL_0006
  IL_0012:  ldloc.0
  IL_0013:  call
    void [mscorlib]Console::WriteLine(int32)
```

```
  IL_0018:  ret
} // end of method Simple::Main
```

that becomes in Pentium assembly language:

```
00000000  push        ebp
00000001  mov         ebp,esp
00000003  sub         esp,8
00000006  push        edi
00000007  push        esi
00000008  xor         esi,esi
0000000a  xor         edi,edi
0000000c  xor         esi,esi
;             for(int i = 0; i < 5; i++){
0000000e  xor         edi,edi
00000010  nop
00000011  jmp         00000016
;                   sum += i;
00000013  add         esi,edi
;             for(int i = 0; i < 5; i++){
00000015  inc         edi
00000016  cmp         edi,5
00000019  jl          00000013
;             Console.WriteLine(sum);
0000001b  mov         ecx,esi
0000001d  call        dword ptr ds:[042125C8h]
;         }
00000023  nop
00000024  pop         esi
00000025  pop         edi
00000026  mov         esp,ebp
00000028  pop         ebp
00000029  ret
```

Security restrictions are implemented at this level in order to make it extremely difficult to bypass. To seamlessly bypass security would require replacing the CLR with a hacked CLR, not impossible to conceive, but hopefully beyond the range of script kiddies and requiring an administration-level compromise from which to start. The security model of .NET consists of checks that occur at both the moment the class is loaded into memory and at the moment that possibly-restricted operations are requested.

Although CIL is not representative of any real machine code, it is *not* interpreted. After the CIL of a class is loaded into memory, as methods in the class are executed, a Just-In-Time compiler (JIT) transforms it from CIL into machine language appropriate to the native CPU. One interesting benefit of this is that it's conceivable that different JIT compilers might become available for different CPUs within a general family (thus, we might eventually have an Itanium JIT, a Pentium JIT, an Athlon JIT, etc.).

The CLR contains a subsystem responsible for memory management inside what is called "managed code." In addition to garbage collection (the process of recycling memory), the CLR memory manager defragments memory and decreases the span of reference of in-memory references (both of which are beneficial side effects of the garbage collection architecture).

Finally, all programs require some basic execution support at the level of thread scheduling, code execution, and other system services. Once again, at this low level, all of this support can be shared by any .NET application, no matter what the originating programming language.

The Common Language Runtime, the base framework classes within mscorlib.dll, and the C# language were submitted by Microsoft to the European Computer Manufacturers Association (ECMA) were ratified as standards in late 2001; in late 2002, a subcommittee of the International Organization for Standardization cleared the way for similar ratification by ISO. The Mono Project (www.go-mono.com) is an effort to create an Open Source implementation of these standards that includes Linux support.

Comments and embedded documentation

There are two types of comments in C#. The first is the traditional C-style comment that was inherited by C++. These comments begin with a /* and continue, possibly across many lines, until a */. Note that many programmers will begin each line of a continued comment with a *, so you'll often see:

```
/* This is a comment
*  that continues
*  across lines
*/
```

Remember, however, that everything inside the /* and */ is ignored, so there's no difference in saying:

```
/* This is a comment that
continues across lines */
```

The second form of comment also comes from C++. It is the single-line comment, which starts at a // and continues until the end of the line. This type of comment is convenient and commonly used because it's easy. You don't need to hunt on the keyboard to find / and then * (instead, you just press the same key twice), and you don't need to close the comment. So you will often see:

```
// this is a one-line comment
```

Documentation Comments

One of the thoughtful parts of the C# language is that the designers didn't consider writing code to be the only important activity—they also thought about documenting it. Possibly the biggest problem with documenting code has been maintaining that documentation. If the documentation and the code are separate, it becomes a hassle to change the documentation every time you change the code. The solution seems simple: link the code to the documentation. The easiest way to do this is to put everything in the same file. To complete the picture, however, you need a special comment syntax to mark special documentation, and a tool to extract those comments and put them in a useful form. This is what C# has done.

Comments that begin with /// can be extracted from source files by running **csc /doc:OutputFile.XML**. Inside the comments you can place any valid XML tags including some tags with predefined meanings discussed next. The resulting XML file is interpreted in certain ways inside of Visual Studio .NET or can be styled with XSLT to produce a Web page or printable documentation. If you don't understand XML, don't worry about it; you'll become much more familiar with it in Chapter 14!

If you run

```
csc /doc:HelloDate.xml HelloDate.cs
```

The resulting XML will be:

```
<?xml version="1.0"?>
<doc>
  <assembly>
        <name>HelloDate</name>
    </assembly>
    <members>
    </members>
</doc>
```

The XML consists of a "doc" element, which is for the assembly named "HelloDate" and which doesn't have any documentation comments.

Tag	Suggested Use
<summary> </summary>	A brief overview of the code element
<remarks> </remarks>	This is used for a more comprehensive discussion of the element's intended behavior.
<param name="name"> </param>	One of these tags should be written for each argument to a method; the value of the **name** attribute specifies which argument. The description should include any preconditions associated with the argument. Preconditions are what the method requires of its arguments so that the method can function correctly. For instance, a precondition of a square root function might be that the input integer be positive.
<returns> </returns>	Methods that return anything other than void should have one of these tags. The contents of the tag should describe what about the return value can be guaranteed. Can it be null? Does it always fall within a certain range? Is it always in a certain state? etc.
<exception cref="type"> </exception>	Every exception that is explicitly raised within the method's body should be documented in a tag such as this (the type of the exception should be the value of the **cref** attribute). To the extent possible, the circumstances which give rise to the exception being thrown should be detailed. Because of C#'s exception model (discussed in Chapter 11), special attention should be paid to making sure that these comments are consistently and uniformly written and maintained.
<permission cref="type"> </permission>	This tag describes the security permissions that are required for the type. The **cref** attribute is optional, but if it exists, it should refer to a PermissionSet associated with the type.
<example> <c></c> <code></code>	The <example> tag should contain a description of a sample use of the code element. The <c> tag is intended to specify an inline code element while the <code> tag is intended for

</example>	multiline snippets.
<see cref="other"> </see> <seealso cref="other"> </seealso>	These tags are intended for cross references to other code elements or other documentation fragments. The <see> tag is intended for inline cross-references, while the <seealso> tag is intended to be broken out into a separate "See Also" section.
<value> </value>	Every externally visible property within a class should be documented with this tag.
<paramref name="arg"/>	This empty tag is used when commenting a method to indicate that the value of the **name** attribute is actually the name of one of the method's arguments.
<list type= [bullet | number | table]> <listheader> <term></term> <description> </description> </listheader> <item> <term></term> <description> </description> </item> </list>	Intended to provide a hint to the XML styler on how to generate documentation.
<para></para>	Intended to specify separate paragraphs within a description or other lengthy text block

Documentation example

Here's the HelloDate C# program, this time with documentation comments added:

```
//:c03:HelloDate2.cs
using System;

namespace ThinkingIn.CSharp.Chap03{
```

```
  ///<summary>Shows doc comments</summary>
  ///<remarks>The documentation comments within C#
  ///are remarkably useful, both within the Visual
  ///Studio environment and as the basis for more
  ///significant printed documentation</remarks>
  public class HelloDate2 {

    ///<summary>Entry point</summary>
    ///<remarks>Prints greeting to
    /// <paramref name="args[0]"/>, gets a
    /// <see cref="System.DateTime">DateTime</see>
    /// and subsequently prints it</remarks>
    ///<param name="args">Command-line should have a
    ///single name. All other args will be ignored
    ///</param>
    public static void Main(string[] args) {
      Console.WriteLine("Hello, {0} it's: ", args[0]);
      Console.WriteLine(DateTime.Now);
    }
  }
}///:~
```

When **csc** extracts the data, it is in this form:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>HelloDate</name>
  </assembly>
  <members>
    <member
      name="T:ThinkingIn.CSharp.Chap03.HelloDate2">
      <summary>Shows doc comments</summary>
      <remarks>The documentation comments within C#
      are remarkably useful, both within the Visual
      Studio environment and as the basis for more
      significant printed documentation</remarks>
    </member>
    <member
name="M:ThinkingIn.CSharp.Chap03.HelloDate2.Main(System.Str
ing[])">
      <summary>Entry point</summary>
```

```
      <remarks>Prints greeting to
       <paramref name="args[0]"/>, gets a
       <see cref="T:System.DateTime">DateTime</see>
       and subsequently prints it</remarks>
      <param name="args">Command-line should have a
      single name. All other args will be ignored
      </param>
    </member>
  </members>
</doc>
```

The first line of the **HelloDate2.cs** file uses a convention that will be used throughout the book. Every compilable sample begins with a comment followed by a ':' the chapter number, another colon, and the name of the file that the example should be saved to. The last line also finishes with a comment, and this one indicates the end of the source code listing, which allows it to be automatically extracted from the text of this book and checked with a compiler. This convention supports a tool which can automatically extract and compile code directly from the "source" Word document.

Coding style

The unofficial standard in C# is to capitalize the first letter of all publicly visible code elements except for parameters. If the element name consists of several words, they are run together (that is, you don't use underscores to separate the names), and the first letter of each embedded word is capitalized, such as:

```
class AllTheColorsOfTheRainbow { // ...
```

This same style is also used for the parts of the class which are intended to be referred to by others (method names and properties). For internal parts fields (member variables) and object reference names, the accepted style is just as it is for classes *except* that the first letter of the identifier is lowercase. For example:

```
class AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  public void ChangeTheHueOfTheColor(int newHue) {
    // ...
  }
  // ...
}
```

Of course, you should remember that the user must also type all these long names, so be merciful. Names, whitespace, and the amount of commenting in a

listing are an area where book authors must follow the dictates of paper cost and tight margins, so please forgive those situations when the listings in this book don't always follow our own guidelines for clarity.

Summary

In this chapter you have seen enough of C# programming to understand how to write a simple program, and you have gotten an overview of the language and some of its basic ideas. However, the examples so far have all been of the form "do this, then do that, then do something else." What if you want the program to make choices, such as "if the result of doing this is red, do that; if not, then do something else"? The support in C# for this fundamental programming activity will be covered in the next chapter.

Exercises

1. Following the **HelloDate.cs** example in this chapter, create a "hello, world" program that simply prints out that statement. You need only a single method in your class (the "Main" one that gets executed when the program starts). Remember to make it **static**. Compile the program with **csc** and run it from the command-line.

2. Find the code fragments involving **ATypeName** and turn them into a program that compiles and runs.

3. Turn the **DataOnly** code fragments into a program that compiles and runs.

4. Modify Exercise 3 so that the values of the data in **DataOnly** are assigned to and printed in **Main( )**.

5. Write a program that includes and calls the **Storage( )** method defined as a code fragment in this chapter.

6. Turn the sample code that defines the **BluffingStrategy** delegate and use the method **SweetPete.SmilePleasantly( )** to instantiate the delegate into a program that compiles and runs.

7. Create a program that defines a **Coin** enumeration as described in the text and adds up a variety of coin types.

8. Write a program that performs multiplication using the **ImaginaryNumber** struct defined in the text.

9. Turn the **StaticFun** code fragments into a working program.

10. Write a program that prints three arguments taken from the command line. To do this, you'll need to index into the command-line array of **string**s, using the **static void Main(string[] args)** form for your entry point.

11. Turn the **AllTheColorsOfTheRainbow** example into a program that compiles and runs.

12. Find the code for the second version of **HelloDate.cs**, which is the simple comment documentation example. Execute **csc /doc** on the file and view the results with your XML-aware Web browser.

13. Add an HTML list of items to the documentation in Exercise 12.

14. Take the program in Exercise 1 and add comment documentation to it. Extract this comment documentation and view it with your Web browser.

15. You have been approached by an android manufacturer to develop the control system for a robotic servant. Describe a party in object-oriented terms. Use abstractions such as **Food** so that you can encompass the entire range of data and behavior between drawing up the invitation list to cleaning up the house afterward.

16. Take the **Food** abstraction from Exercise 15 and describe it more fully in terms of classes and types. Use inheritance in at least two places. Constrain your model to the data and behaviors appropriate to the robotic butler.

17. Choose one of the classes developed in Exercise 16 that requires some complex behavior (perhaps an item that needs baking or the purchase of exotic ingredients). List the classes that would be required to collaborate to accomplish the complex behavior. For instance, if the behavior was lighting candles on a cake, the classes might include **Candle**, **Cake**, and **Match**.

4: Controlling Program Flow

Like a sentient creature, a program must manipulate its world and make choices during execution.

In C# you manipulate objects and data using operators, and you make choices with execution control statements. The statements used will be familiar to programmers with Java, C++, or C backgrounds, but there are a few that may seem unusual to programmers coming from Visual Basic backgrounds.

Using C#operators

An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary method calls, but the effect is the same. You should be reasonably comfortable with the general concept of operators from your previous programming experience. Addition (+), subtraction and unary minus (-), multiplication (*), division (/), and assignment (=) all work much the same in any programming language.

All operators produce a value from their operands. In addition, an operator can change the value of an operand. This is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but you should keep in mind that the value produced is available for your use just as in operators without side effects.

Operators work with all primitives and many objects. When you program your own objects, you will be able to extend them to support whichever primitives make sense (you'll find yourself creating '+' operations far more often than '/' operations!) The operators '**=**', '==' and '**!**=', work for all objects and are a point of confusion for objects that we'll deal with in #reference#.

Precedence

Operator precedence defines how an expression evaluates when several operators are present. C# has specific rules that determine the order of evaluation. The easiest one to remember is that multiplication and division happen before

addition and subtraction. Programmers often forget the other precedence rules, so you should use parentheses to make the order of evaluation explicit. For example:

```
a = x + y - 2/2 + z;
```

has a very different meaning from the same statement with a particular grouping of parentheses:

```
a = x + (y - 2)/(2 + z);
```

Assignment

Assignment is performed with the operator =. It means "take the value of the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*). An rvalue is any constant, variable or expression that can produce a value, but an lvalue must be a distinct, named variable. (That is, there must be a physical space to store a value.) For instance, you can assign a constant value to a variable (**A = 4;**), but you cannot assign anything to constant value—it cannot be an lvalue. (You can't say **4 = A;**.)

Assignment of primitives is quite straightforward. Since the primitive holds the actual value and not a reference to an object, when you assign primitives you copy the contents from one place to another. For example, if you say **A = B** for primitives, then the contents of **B** are copied into **A**. If you then go on to modify **A**, **B** is naturally unaffected by this modification. As a programmer, this is what you've come to expect for most situations.

When you assign objects, however, things change. Whenever you manipulate an object, what you're manipulating is the reference, so when you assign "from one object to another" you're actually copying a reference from one place to another. This means that if you say **C = D** for objects, you end up with both **C** and **D** pointing to the object that, originally, only **D** pointed to. The following example will demonstrate this.

Here's the example:

```
//:c03:Assignment.cs
using System;

class Number {
  public int i;
}

public class Assignment {
```

```
  public static void Main(){
    Number n1 = new Number();
    Number n2 = new Number();
    n1.i = 9;
    n2.i = 47;
    Console.WriteLine(
      "1: n1.i: " + n1.i + ", n2.i: " + n2.i);
    n1 = n2;
    Console.WriteLine(
      "2: n1.i: " + n1.i + ", n2.i: " + n2.i);
    n1.i = 27;
    Console.WriteLine(
      "3: n1.i: " + n1.i + ", n2.i: " + n2.i);
  }
}///:~
```

The **Number** class is simple, and two instances of it (**n1** and **n2**) are created within M**ain( )**. The **i** value within each **Number** is given a different value, and then **n2** is assigned to **n1**, and **n1** is changed. In many programming languages you would expect **n1** and **n2** to be independent at all times, but because you've assigned a reference here's the output you'll see:

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

Changing the **n1** object appears to change the **n2** object as well! This is because both **n1** and **n2** contain the same reference, which is pointing to the same object. (The original reference that was in **n1** that pointed to the object holding a value of 9 was overwritten during the assignment and effectively lost; its object will be cleaned up by the garbage collector.)

This phenomenon is called *aliasing* and it's a fundamental way that C# works with objects. But what if you don't want aliasing to occur in this case? You could forego the assignment and say:

```
n1.i = n2.i;
```

This retains the two separate objects instead of tossing one and tying **n1** and **n2** to the same object, but you'll soon realize that manipulating the fields within objects is messy and goes against good object-oriented design principles.

Aliasing during method calls

Aliasing will also occur when you pass an object into a method:

```
//:c04:PassObject.cs
using System;
class Letter {
  public char c;
}

public class PassObject {
  static void f(Letter y){
    y.c = 'z';
  }

  public static void Main(){
    Letter x = new Letter();
    x.c = 'a';
    Console.WriteLine("1: x.c: " + x.c);
    f(x);
    Console.WriteLine("2: x.c: " + x.c);
  }
}///:~
```

In many programming languages, the method **F( )** would appear to be making a copy of its argument **Letter y** inside the scope of the method. But once again a reference is being passed so the line

```
y.c = 'z';
```

is actually changing the object outside of **F( )**. The output shows this:

```
1: x.c: a
2: x.c: z
```

Aliasing and object state

Methods actually receive *copies* of their arguments, but since a copy of a reference points to the same thing as the original, aliasing occurs. In this example, **Viewer** objects fight over control of a television set. Although each viewer receives a copy of the reference to the **Television**, when they change the state of the **Television**, everyone has to live with the results:

```
//:c04:ChannelBattle.cs
//Shows aliasing in method calls
using System;

class Television {
```

```
  int channel = 2;
  internal int Channel{
    get { return channel;}
    set {
      Console.WriteLine("Everyone sees {0}", value);
      channel = value;
    }
  }
}

class Viewer {
  static Random rand = new Random();
  int preferredChannel = rand.Next(13);

  static int counter = 0;
  int viewerId = counter++;

  void ChangeChannel(Television tv){
    Console.WriteLine(
      "Viewer {0} doesn't like {1}, switch to {2}",
      viewerId, tv.Channel, preferredChannel);
    tv.Channel = preferredChannel;
  }

  public static void Main(){
    Viewer v0 = new Viewer();
    Viewer v1 = new Viewer();
    Viewer v2 = new Viewer();
    Television tv = new Television();
    v0.ChangeChannel(tv);
    v1.ChangeChannel(tv);
    v2.ChangeChannel(tv);
  }
}///:~
```

The **Television** object has a property called **Channel**. The **int channel** represents the **Television** object's *state*. Everyone watching that particular **Television** watches the same channel; all references to a particular object are dependent on that object's state.

A **Viewer** object has an **int** value that is the **preferredChannel**. A particular viewer's **preferredChosen** is determined randomly by a **Random** object that

is **static** and therefore shared by all **Viewer**s (as described in Chapter 2). Similarly, there is a **static int counter** that is shared by all **Viewer**s and an **int viewerId** that is particular to an individual. As **static** variables, **rand** and **counter** can be said to contribute to the class's *shared state*, while **preferredChannel** and **viewerId** determine the **Viewer**'s object's *state* (more accurately called the *object state* or *instance state* to distinguish it from the class's shared state).

The **Viewer.Main( )** method creates 3 **Viewer** objects. Before the first **Viewer** is created, the **Viewer** class state is initialized, setting the **counter** variable to zero. Every time a **Viewer** is created, it sets its **viewerId** variable to the value of the **counter** and increments the **counter**; the object state of each **Viewer** reads from and then modifies the class state of the **Viewer** type.

After the **Viewer**s have been created, we create a single **Television** object, which when it's created is tuned to Channel 2. A reference to that **Television** object is handed to each of the **Viewer**s in turn by way of a call to **Viewer.ChangeChannel( )**. Although each viewer receives a *copy of the reference* to the **Television**, the copy always points to the same **Television**. Everyone ends up watching the same channel as the state of the **Television** is manipulated.

One of the cardinal rules of object-oriented programming is to distribute state among objects. It is possible to imagine storing the current channel being watched as a static variable in the **Viewer** class or for the **Television** to keep a list of **Viewer**s and their preferred channels. But when programming (and especially when changing a program you haven't seen in a while) often the hardest thing is knowing the precise state that your class is in when a particular line is executed. Generally, it's easier to modify classes that don't have complex *state transitions*.

Aliasing and the ref keyword

Since object-oriented programming is mostly concerned with objects, and objects are always manipulated by references, the fact that methods are passed copies of their arguments doesn't matter: a copy of a reference refers to the same thing as the original reference. However, with C#'s value types, such as primitive number types, **struct**s, and **enum**s, it matters a lot. This program is almost identical to the previous example, but this time we have an **Mp3Player** defined not as a class, but as a **struct**.

```
//:c04:Mp3Player.cs
//Demonstrates value types dont alias
```

```
using System;

struct Mp3Player {
  int volume;
  internal int Volume{
    get { return volume;}
    set {
      volume = value;
      Console.WriteLine(
        "Volume set to {0} ", volume);
    }
  }
}

class Viewer {
  static Random rand = new Random();
  int preferredVolume = rand.Next(10);

  static int counter = 0;
  int viewerId = counter++;

  void ChangeVolume(Mp3Player p){
    Console.WriteLine(
      "Viewer {0} doesn't like {1}, switch to {2}",
      viewerId, p.Volume, preferredVolume);
    p.Volume = preferredVolume;
  }

  public static void Main(){
    Viewer v0 = new Viewer();
    Viewer v1 = new Viewer();
    Viewer v2 = new Viewer();
    Mp3Player p = new Mp3Player();
    v0.ChangeVolume(p);
    v1.ChangeVolume(p);
    v2.ChangeVolume(p);
  }
}///:~
```

Mp3Player is a value type, so when **Viewer.ChangeVolume( )** receives a copy (as is normally the case with arguments), the state of the *copy* is manipulated, not the state of the original **Mp3Player**. Every **Viewer** receives a

copy of the **Mp3Player**'s original state, with the volume at zero. The output of the program is:

```
Viewer 0 doesn't like 0, switch to 6
Volume set to 6
Viewer 1 doesn't like 0, switch to 0
Volume set to 0
Viewer 2 doesn't like 0, switch to 5
Volume set to 5
```

C#'s **ref** keyword passes, not a copy of the argument, but a reference to the argument. If the argument is itself a reference (as when the variable is referencing an object), the reference to the reference still ends up manipulating the same object. But when the argument is a value type, it makes a lot of difference. To use the **ref** keyword, you must add it to both the argument list inside the method you are creating as well as use it as a prefix during the call. Here's the above example, with **ref** added:

```
//:c04:Mp3Player2.cs
//Demonstrates value types dont alias
using System;

struct Mp3Player {
  int volume;
  internal int Volume{
    get { return volume;}
    set {
      volume = value;
      Console.WriteLine(
        "Volume set to {0} ", volume);
    }
  }
}

class Viewer {
  static Random rand = new Random();
  int preferredVolume = rand.Next(10);

  static int counter = 0;
  int viewerId = counter++;

  void ChangeVolume(ref Mp3Player p){
```

```
    Console.WriteLine(
      "Viewer {0} doesn't like {1}, switch to {2}",
      viewerId, p.Volume, preferredVolume);
    p.Volume = preferredVolume;
  }

  public static void Main(){
    Viewer v0 = new Viewer();
    Viewer v1 = new Viewer();
    Viewer v2 = new Viewer();
    Mp3Player p = new Mp3Player();
    v0.ChangeVolume(ref p);
    v1.ChangeVolume(ref p);
    v2.ChangeVolume(ref p);
  }
}///:~
```

The changes are in the lines:

```
void ChangeVolume(ref Mp3Player p){ … }
…
v0.ChangeVolume(ref p);
```

Now when run, each **Viewer** receives a reference to the original **Mp3Player**, whose state changes from call to call:

```
Viewer 0 doesn't like 0, switch to 1
Volume set to 1
Viewer 1 doesn't like 1, switch to 7
Volume set to 7
Viewer 2 doesn't like 7, switch to 4
Volume set to 4
```

Beyond aliasing with out

Usually, when you calling a method that will manipulate the state of objects, you have references to preexisting objects and you rely on aliasing. If you need to create a new object inside a method, the preferred way of returning a reference to it for use in the outside world is to return it as the method's **return** value:

```
Sandwich MakeASandwich(Bread slice1, Bread slice2,
   Meat chosenMeat, Lettuce lettuce){
   Sandwich s = new Sandwich();
   s.TopSlice = slice1;
```

```
    s.BottomSlice = slice2;
    …etc…
    return s;
}
```

However, if you need a method that returns more than one object (which is rare, since a method should do one thing), and you can't initialize the objects before the call, you can use C#'s **out** keyword. Usually, C#'s compiler will not allow you to use references that you have declared but not initialized. The **out** keyword, though, tells the compiler that the initialization of those variables is the responsibility of the called method.

To use **out**, you put it in the argument list of the method and prefix the reference in the actual call.

```
//:c04:BreadDissector.cs
using System;

class Bread {
}
class Meat {
}
class Lettuce {
}

class Sandwich {
  internal Sandwich(){
    topSlice = new Bread();
    bottomSlice = new Bread();
    meat = new Meat();
    lettuce = new Lettuce();
  }

  Bread topSlice, bottomSlice;
  internal Bread TopSlice{
    get { return topSlice;}
  }
  internal Bread BottomSlice{
    get { return bottomSlice;}
  }

  Meat meat;
```

```
  internal Meat Meat{
    get { return meat;}
  }
  Lettuce lettuce;
  internal Lettuce Lettuce{
    get { return lettuce;}
  }
}

class Dissector {
  void Split(
    Sandwich s, out Bread s1, out Bread s2,
    out Meat m, out Lettuce l){
    s1 = s.TopSlice;
    s2 = s.BottomSlice;
    m = s.Meat;
    l = s.Lettuce;
  }

  public static void Main(){
    Sandwich s = new Sandwich();
    Bread b1, b2;
    Meat m;
    Lettuce l;
    Dissector d = new Dissector();
    d.Split(s, out b1, out b2, out m, out l);
    Console.WriteLine(
      "{0} {1} {2} {3}", b1, b2, m, l, b2);
  }
}///:~
```

The **Sandwich** class constructs its constituent components (two pieces of **Bread**, some **Meat**, and some **Lettuce**) during the **Sandwich( )** constructor call. Each of these components is available in a property field of the **Sandwich** and that's normally how you'd get them, one at a time. However, the **Dissector.Split( )** method might be more convenient in some circumstances. Although the **Dissector.Split( )** method itself accesses the components one by one, *all* of the arguments marked with **out** are initialized within **Dissector.Split( )**. The **Dissector.Main( )** declares and initializes a **Sandwich** but just declares references to the components. If **Dissector.Main( )** did *not* call **Dissector.Split( )**, the compiler would not

allow the last line of **Dissector.Main( )**, saying that the variables **b1, b2, m,** and **l** were not initialized. The line

```
d.Split(s, out b1, out b2, out m, out l);
```

tells the compiler to delegate the initialization responsibility to **Dissector.Split( )**. Since **Dissector.Split( )** fulfills the initialization responsibility, the method runs fine, "returning" four objects.

Mathematical operators

The basic mathematical operators are the same as the ones available in most programming languages: addition (**+**), subtraction (**-**), division (**/**), multiplication (*****) and modulus (**%**, which produces the remainder from integer division). Integer division truncates, rather than rounds, the result.

C# also uses a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable **x** and assign the result to **x**, use: **x += 4**.

This example shows the use of the mathematical operators:

```
//:c04:MathOps.cs
using System;

public class MathOps {
  ///Prints a string and an int:
  static void PInt(String s, int i){
    Console.WriteLine(s + " = " + i);
  }

  //Shorthand to print a string and a float
  static void PDouble(String s, double f){
    Console.WriteLine(s + " = " + f);
  }

  public static void Main(){
    //Create a random number generator,
    //seeds with current time by default
    Random rand = new Random();
    int i, j, k;
    //get a positive random number less than
    //the specified maximum
```

```
    j = rand.Next(100);
    k = rand.Next(100);
    PInt("j",j); PInt("k",k);
    i = j + k; PInt("j + k", i);
    i = j - k; PInt("j - k", i);
    i = k / j; PInt("k / j", i);
    i = k * j; PInt("k * j", i);
    //Limits i to a positive number less than j
    i = k % j; PInt("k % j", i);
    j %= k; PInt("j %= k", j);
    //Floating-point number tests:
    double u, v, w;
    v = rand.NextDouble();
    w= rand.NextDouble();
    PDouble("v", v); PDouble("w",w);
    u = v + w; PDouble("v + w", u);
    u = v - w; PDouble("v - w", u);
    u = v * w; PDouble("v * w", u);
    u = v / w; PDouble("v / w", u);
    //the following also works for
    //char, byte, short, int, long,
    //and float
    u += v; PDouble("u += v", u);
    u -= v; PDouble("u -= v", u);
    u *= v; PDouble("u *= v", u);
    u /= v; PDouble("u /= v", u);
  }
}///:~
```

The first thing you will see are some shorthand methods for printing: the **Prt( )** method prints a **String**, the **PInt( )** prints a **String** followed by an **int** and the **PDouble( )** prints a **String** followed by a **double**. Of course, they all ultimately end up using **Console.WriteLine( )**, but these methods are slightly more space-efficient for the cramped margin of a book.

To generate numbers, the program first creates a **Random** object. Because no arguments are passed during creation, C# uses the current time as a seed for the random number generator. The program generates a number of different types of random numbers with the **Random** object simply by calling the methods: **Next ( )** and **NextDouble( )** (you can also call N**extLong( )** or **Next(int)**).

Unary minus and plus operators

The unary minus (-) and unary plus (+) are the same operators as binary minus and plus. The compiler figures out which use is intended by the way you write the expression. For instance, the statement

```
x = -a;
```

has an obvious meaning. The compiler is able to figure out:

```
x = a * -b;
```

but the reader might get confused, so it is clearer to say:

```
x = a * (-b);
```

The unary minus produces the negative of the value. Unary plus provides symmetry with unary minus, although it doesn't have any effect.

Auto increment and decrement

C#, like C, is full of shortcuts. Shortcuts can make code much easier to type, and either easier or harder to read.

Two of the nicer shortcuts are the increment and decrement operators (often referred to as the auto-increment and auto-decrement operators). The decrement operator is **--** and means "decrease by one unit." The increment operator is ++ and means "increase by one unit." If **a** is an **int**, for example, the expression **++a** is equivalent to (**a = a + 1**). Increment and decrement operators produce the value of the variable as a result.

There are two versions of each type of operator, often called the prefix and postfix versions. Pre-increment means the ++ operator appears before the variable or expression, and post-increment means the ++ operator appears after the variable or expression. Similarly, pre-decrement means the **--** operator appears before the variable or expression, and post-decrement means the **--** operator appears after the variable or expression. For pre-increment and pre-decrement, (i.e., **++a** or **--a**), the operation is performed and the value is produced. For post-increment and post-decrement (i.e. **a++** or **a--**), the value is produced, then the operation is performed. As an example:

```
//:c04:AutoInc.cs
using System;

public class AutoInc {
  public static void Main(){
```

```
    int i = 1;
    prt("i: " + i);
    prt("++i: " + ++i); //Pre-increment
    prt("i++: " + i++); //Post-increment
    prt("i: " + i);
    prt("--i: " + --i); //Pre-increment
    prt("i--: " + i--); //Post-increment
    prt("i: " + i);
  }

  static void prt(String s){
    Console.WriteLine(s);
  }
}///:~
```

The output for this program is:

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

You can see that for the prefix form you get the value after the operation has been performed, but with the postfix form you get the value before the operation is performed. These are the only operators (other than those involving assignment) that have side effects. (That is, they change the operand rather than using just its value.)

The increment operator is one explanation for the name C++, implying "one step beyond C." As for C#, the explanation seems to be in music, where the # symbolizes "sharp" – a half-step "up[1]."

Relational operators

Relational operators generate a boolean result. They evaluate the relationship between the values of the operands. A relational expression produces **true** if the relationship is true, and **false** if the relationship is untrue. The relational

---

[1] Michael Lamsoul has wittily suggested that the # in C# may also be a geometric pun on the ++ in C++, that the sharp looks like a square of + operators.

operators are less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), equivalent (==) and not equivalent (!=). Equivalence and nonequivalence work with all built-in data types, but the other comparisons won't work with type **bool**.

Testing object equivalence

The relational operators == and **!=** also work with all objects, but their meaning often confuses the first-time C# programmer. Here's an example:

```
//:c04:EqualsOperator.cs
using System;
class MyInt {
  Int32 i;
  public MyInt(int j){
    i = j;
  }
}
//Demonstrates handle inequivalence.
public class EqualsOperator {
  public static void Main(){
    MyInt m1 = new MyInt(47);
    MyInt m2 = new MyInt(47);
    Console.WriteLine("m1 == m2: "
      + (m1 == m2));
  }
}///:~
```

The expression **System.Console.WriteLine(m1 == m2)** will print the result of the **bool** comparison within it. Surely the output should be **true**, since both **MyInt** objects have the same value. But while the *contents* of the objects are the same, the references are not the same and the operators == and **!=** compare object references. So the output is actually **false**. Naturally, this surprises people at first.

What if you want to compare the actual contents of an object for equivalence? For objects in a well-designed class library (such as the .NET framework), you just use the equivalence operator == that has been specially *overridden* in many classes to get the desired behavior. Unfortunately, you won't learn about overriding until Chapter 7, but being aware of the way '==' behaves might save you some grief in the meantime.

Logical operators

Each of the logical operators AND (&&), OR (||) and NOT (!) produces a **bool** value of **true** or **false** based on the logical relationship of its arguments. This example uses the relational and logical operators:

```
//:c04:Bool.cs
using System;
// Relational and logical operators.
public class Bool {
  public static void Main() {
    Random rand = new Random();
    int i = rand.Next(100);
    int j = rand.Next(100);
    Prt("i = " + i);
    Prt("j = " + j);
    Prt("i > j is " + (i > j));
    Prt("i < j is " + (i < j));
    Prt("i >= j is " + (i >= j));
    Prt("i <= j is " + (i <= j));
    Prt("i == j is " + (i == j));
    Prt("i != j is " + (i != j));

    // Treating an int as a boolean is
    // not legal C#
    //! Prt("i && j is " + (i && j));
    //! Prt("i || j is " + (i || j));
    //! Prt("!i is " + !i);

    Prt("(i < 10) && (j < 10) is "
      + ((i < 10) && (j < 10)) );
    Prt("(i < 10) || (j < 10) is "
      + ((i < 10) || (j < 10)) );
  }
  static void Prt(String s) {
    Console.WriteLine(s);
  }
}///:~
```

You can apply AND, OR, or NOT to **bool** values only. You can't use a non-**bool** as if it were a **bool** in a logical expression as you can in some other languages. You can see the failed attempts at doing this commented out with a **//!** comment

marker. The subsequent expressions, however, produce **bool** values using relational comparisons, then use logical operations on the results.

One output listing looked like this:

```
i = 85
j = 4
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true
```

Note that a **bool** value is automatically converted to an appropriate text form if it's appended to a **string**.

You can replace the definition for **int** in the above program with any other primitive data type except **bool**. Be aware, however, that the comparison of floating-point numbers is very strict. A number that is the tiniest fraction different from another number is still "not equal." A number that is the tiniest bit above zero is still nonzero.

Short-circuiting

When dealing with logical operators you run into a phenomenon called "short circuiting." This means that the expression will be evaluated only *until* the truth or falsehood of the entire expression can be unambiguously determined. As a result, all the parts of a logical expression might not be evaluated. Here's an example that demonstrates short-circuiting:

```
//:c04:ShortCircuit.cs
// Demonstrates short-circuiting behavior.
// with logical operators.
using System;

public class ShortCircuit {
  static bool Test1(int val) {
    Console.WriteLine("Test1(" + val + ")");
    Console.WriteLine("result: " + (val < 1));
    return val < 1;
  }
```

```
  static bool Test2(int val) {
    Console.WriteLine("Test2(" + val + ")");
    Console.WriteLine("result: " + (val < 2));
    return val < 2;
  }
  static bool Test3(int val) {
    Console.WriteLine("Test3(" + val + ")");
    Console.WriteLine("result: " + (val < 3));
    return val < 3;
  }
  public static void Main() {
    if (Test1(0) && Test2(2) && Test3(2))
      Console.WriteLine("expression is true");
    else
      Console.WriteLine("expression is false");
  }
} ///:~
```

Each test performs a comparison against the argument and returns true or false. It also prints information to show you that it's being called. The tests are used in the expression:

```
if(test1(0) && test2(2) && test3(2))
```

You might naturally think that all three tests would be executed, but the output shows otherwise:

```
Ttest1(0)
result: true
Ttest2(2)
result: false
expression is false
```

The first test produced a **true** result, so the expression evaluation continues. However, the second test produced a **false** result. Since this means that the whole expression must be **false**, why continue evaluating the rest of the expression? It could be expensive. The reason for short-circuiting, in fact, is precisely that; you can get a potential performance increase if all the parts of a logical expression do not need to be evaluated.

Bitwise operators

There are only 10 types of people in this world: those that understand binary and those that don't. C#'s bitwise operators are for those that do. You use the bitwise

operators to manipulate individual bits in an integral primitive data type. Bitwise operators perform boolean algebra on the corresponding bits in the two arguments to produce the result.

The bitwise operators come from C's low-level orientation; you were often manipulating hardware directly and had to set the bits in hardware registers. Although most application and Web Service developers will not be using the bitwise operators much, developers for PocketPCs, set-top boxes, and the XBox often need every bit-twiddling advantage they can get.

The bitwise AND operator (**&**) produces a one in the output bit if both input bits are one; otherwise it produces a zero. The bitwise OR operator (|) produces a one in the output bit if either input bit is a one and produces a zero only if both input bits are zero. The bitwise EXCLUSIVE OR, or XOR (^), produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise NOT (~, also called the *ones complement* operator) is a unary operator; it takes only one argument. (All other bitwise operators are binary operators.) Bitwise NOT produces the opposite of the input bit—a one if the input bit is zero, a zero if the input bit is one.

The bitwise operators and logical operators use the same characters, so it is helpful to have a mnemonic device to help you remember the meanings: since bits are "small," there is only one character in the bitwise operators.

Bitwise operators can be combined with the = sign to unite the operation and assignment: **&=**, |= and ^= are all legitimate. (Since ~ is a unary operator it cannot be combined with the = sign.)

The **bool** type is treated as a one-bit value so it is somewhat different. You can perform a bitwise AND, OR and XOR, but you can't perform a bitwise NOT (presumably to prevent confusion with the logical NOT). For **bool**s the bitwise operators have the same effect as the logical operators except that they do not short circuit. Also, bitwise operations on **bool**s include an XOR logical operator that is not included under the list of "logical" operators. You're prevented from using **bool**s in shift expressions, described next.

Shift operators

The shift operators also manipulate bits. They can be used solely with primitive, integral types. The left-shift operator (<<) produces the operand to the left of the operator shifted to the left by the number of bits specified after the operator (inserting zeroes at the lower-order bits). The signed right-shift operator (>>) produces the operand to the left of the operator shifted to the right by the number

of bits specified after the operator. The signed right shift >> uses *sign extension*: if the value is positive, zeroes are inserted at the higher-order bits; if the value is negative, ones are inserted at the higher-order bits. (C# does not have unsigned shifts, but thas unsigned datatypes for such situations.)

If you shift a **char**, **byte,** or **short**, it will be promoted to **int** before the shift takes place, and the result will be an **int**. Only the five low-order bits of the right-hand side will be used. This prevents you from shifting more than the number of bits in an **int**. If you're operating on a **long**, you'll get a **long** result. Only the six low-order bits of the right-hand side will be used so you can't shift more than the number of bits in a **long**.

Shifts can be combined with the equal sign (<<= or >>=). The lvalue is replaced by the lvalue shifted by the rvalue.

Here's an example that demonstrates the use of all the operators involving bits:

```
//:c04:BitManipulation.cs
using System;

public class BitManipulation {
  public static void Main() {
    Random rand = new Random();
    int i = rand.Next();
    int j = rand.Next();
    PBinInt("-1", -1);
    PBinInt("+1", +1);
    int maxpos = Int32.MaxValue;
    PBinInt("maxpos", maxpos);
    int maxneg = Int32.MinValue;
    PBinInt("maxneg", maxneg);
    PBinInt("i", i);
    PBinInt("~i", ~i);
    PBinInt("-i", -i);
    PBinInt("j", j);
    PBinInt("i & j", i & j);
    PBinInt("i | j", i | j);
    PBinInt("i ^ j", i ^ j);
    PBinInt("i << 5", i << 5);
    PBinInt("i >> 5", i >> 5);
    PBinInt("(~i) >> 5", (~i) >> 5);
    long l_high_bits = rand.Next();
```

```
    l_high_bits <<= 32;
    long l = l_high_bits + rand.Next();
    long m_high_bits = rand.Next();
    m_high_bits <<=32;
    long m = m_high_bits + rand.Next();
    PBinLong("-1L", -1L);
    PBinLong("+1L", +1L);
    long ll = Int64.MaxValue;
    PBinLong("maxpos", ll);
    long lln = Int64.MinValue;
    PBinLong("maxneg", lln);
    PBinLong("l_high_bits", l_high_bits);
    PBinLong("l", l);
    PBinLong("~l", ~l);
    PBinLong("-l", -l);
    PBinLong("m_high_bits", m_high_bits);
    PBinLong("m", m);
    PBinLong("l & m", l & m);
    PBinLong("l | m", l | m);
    PBinLong("l ^ m", l ^ m);
    PBinLong("l << 5", l << 5);
    PBinLong("l >> 5", l >> 5);
    PBinLong("(~l) >> 5", (~l) >> 5);
  }
  static void PBinInt(String s, int i) {
    Console.WriteLine(
      s + ", int: " + i + ", binary: ");
    Console.Write("   ");
    for (int j = 31; j >=0; j--)
      if (((1 << j) &  i) != 0)
        Console.Write("1");
      else
        Console.Write("0");
    Console.WriteLine();
  }

  static void PBinLong(String s, long l) {
    Console.WriteLine(
      s + ", long: " + l + ", binary: ");
    Console.Write("   ");
    for (int i = 63; i >=0; i--)
```

```
      if (((1L << i) & l) != 0)
        Console.Write("1");
      else
        Console.Write("0");
    Console.WriteLine();
  }
}///:~
```

The two methods at the end, **PBinInt( )** and **PBinLong( )** take an **int** or a **long**, respectively, and print it out in binary format along with a descriptive string. You can ignore the implementation of these for now.

You'll note the use of **Console.Write( )** instead of **Console.WriteLine( )**. The **Write( )** method does not emit a new line, so it allows you to output a line in pieces.

As well as demonstrating the effect of all the bitwise operators for **int** and **long**, this example also shows the minimum, maximum, +1 and -1 values for **int** and **long** so you can see what they look like. Note that the high bit represents the sign: 0 means positive and 1 means negative. The output looks like this:

```
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   00000000000000000000000000000001
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: 1177419330, binary:
   01000110001011011111111001000010
~i, int: -1177419331, binary:
   10111001110100100000000110111101
-i, int: -1177419330, binary:
   10111001110100100000000110111110
j, int: 886693932, binary:
   00110100110110011110000000101100
i & j, int: 67756032, binary:
   00000100000010011110000000000000
i | j, int: 1996357230, binary:
   01110110111111101111111001101110
i ^ j, int: 1928601198, binary:
   01110010111101000001111001101110
```

```
i << 5, int: -977287104, binary:
   11000101101111111100100001000000
i >> 5, int: 36794354, binary:
   00000010001100010110111111110010
(~i) >> 5, int: -36794355, binary:
   11111101110011101001000000001101
-1L, long: -1, binary:

11111111111111111111111111111111111111111111111111111111111
11111
+1L, long: 1, binary:

00000000000000000000000000000000000000000000000000000000000
00001
maxpos, long: 9223372036854775807, binary:

01111111111111111111111111111111111111111111111111111111111
11111
maxneg, long: -9223372036854775808, binary:

10000000000000000000000000000000000000000000000000000000000
00000
l_high_bits, long: 4654972597212020736, binary:

01000000100110011100100011101010000000000000000000000000000
00000
l, long: 4654972598829014295, binary:

01000000100110011100100011101010011000000110000101011101000
10111
~l, long: -4654972598829014296, binary:

10111111011001100011011100010101100111111001111010100010111
01000
-l, long: -4654972598829014295, binary:

10111111011001100011011100010101100111111001111010100010111
01001
m_high_bits, long: 468354230734815232, binary:
```

```
0000011001111111111011011011000100000000000000000000000000000
00000
m, long: 468354231158705547, binary:

0000011001111111111011011011000100011001010001000000110110010
01011
l & m, long: 7257463942286595, binary:

0000000000011001110010001010000000000000100000000000110100000
00011
l | m, long: 5116069366045433247, binary:

0100011011111111111011011111101101110010110010101011101100
11111
l ^ m, long: 5108811902103146652, binary:

0100011011100110001001010101101101110010010010101010000100
11100
l << 5, long: 1385170572852044512, binary:

0001001100111001000111010100110000001100001010111010001011100000
l >> 5, long: 145467893713406696, binary:

0000001000000100110011100100011101010011000000110000101011101000
(~l) >> 5, long: -145467893713406697, binary:

1111110111111011001100011011100010101100111111001111010100010111
```

The binary representation of the numbers is referred to as *signed two's complement*.

Ternary if-else operator

This operator is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary if-else statement that you'll see in the next section of this chapter. The expression is of the form:

```
boolean-exp ? value0 : value1
```

If *boolean-exp* evaluates to **true**, *value0* is evaluated and its result becomes the value produced by the operator. If *boolean-exp* is **false**, *value1* is evaluated and its result becomes the value produced by the operator.

Of course, you could use an ordinary **if-else** statement (described later), but the ternary operator is much terser. Although C (where this operator originated) prides itself on being a terse language, and the ternary operator might have been introduced partly for efficiency, you should be somewhat wary of using it on an everyday basis—it's easy to produce unreadable code.

The conditional operator can be used for its side effects or for the value it produces, but in general you want the value since that's what makes the operator distinct from the **if-else**. Here's an example:

```
static int Ternary(int i) {
  return i < 10 ? i * 100 : i * 10;
}
```

You can see that this code is more compact than what you'd need to write without the ternary operator:

```
static int Alternative(int i) {
  if (i < 10)
    return i * 100;
  else
    return i * 10;
}
```

The second form is easier to understand, and doesn't require a lot more typing. So be sure to ponder your reasons when choosing the ternary operator – it's generally only warranted when you're setting a variable to one of two straightforward values:

```
int ternaryResult = i < 10 ? i * 100 : i * 10;
```

The comma operator

The comma is used in C and C++ not only as a separator in function argument lists, but also as an operator for sequential evaluation. The sole place that the comma *operator* is used in C# is in **for** loops, which will be described later in this chapter.

Common pitfalls when using operators

One of the pitfalls when using operators is trying to get away without parentheses when you are even the least bit uncertain about how an expression will evaluate. This is still true in C#.

An extremely common error in C and C++ looks like this:

```
while(x = y) {
    // ....
}
```

The programmer was trying to test for equivalence (==) rather than do an assignment. In C and C++ the result of this assignment will always be **true** if **y** is nonzero, and you'll probably get an infinite loop. In C#, the result of this expression is not a **bool,** and the compiler expects a **bool** and won't convert from an **int**, so it will conveniently give you a compile-time error and catch the problem before you ever try to run the program. So the pitfall never happens in C#. (The only time you won't get a compile-time error is when **x** and **y** are **bool**, in which case **x = y** is a legal expression, and in the above case, probably an error.)

A similar problem in C and C++ is using bitwise AND and OR instead of the logical versions. Bitwise AND and OR use one of the characters (**&** or **|**) while logical AND and OR use two (**&&** and **||**). Just as with = and ==, it's easy to type just one character instead of two. In C#, the compiler again prevents this because it won't let you cavalierly use one type where it doesn't belong.

Casting operators

The word *cast* is used in the sense of "casting into a mold." C# will automatically change one type of data into another when appropriate. For instance, if you assign an integral value to a floating-point variable, the compiler will automatically convert the **int** to a **float**. Casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

To perform a cast, put the desired data type (including all modifiers) inside parentheses to the left of any value. Here's an example:

```
void Casts() {
  int i = 200;
  long aLongVar = (long)i;
  long anotherLongVar = (long)200;
```

```
}
```

As you can see, it's possible to perform a cast on a numeric value as well as on a variable. In both casts shown here, however, the cast is superfluous, since the compiler will automatically promote an **int** value to a **long** when necessary. However, you are allowed to use superfluous casts to make a point or to make your code more clear. In other situations, a cast may be essential just to get the code to compile.

In C and C++, casting can cause some headaches. In C#, casting is safe, with the exception that when you perform a so-called *narrowing conversion* (that is, when you go from a data type that can hold more information to one that doesn't hold as much) you run the risk of losing information. Here the compiler forces you to do a cast, in effect saying "this can be a dangerous thing to do—if you want me to do it anyway you must make the cast explicit." With a *widening conversion* an explicit cast is not needed because the new type will more than hold the information from the old type so that no information is ever lost.

C# allows you to define casts between any logically exchangeable objects and comes with prewritten casts for the numeric value types. To convert one to the other there must be special methods. (You'll find out later in this book that objects can be cast within a *family* of types without the need to write special casting code; an **Oak** can be cast to a **Tree** and vice-versa, but not to a foreign type such as a **Rock** unless you write an explicit **Tree**-to-**Rock** conversion.)

Literals

Ordinarily when you insert a literal value into a program the compiler knows exactly what type to make it. Sometimes, however, the type is ambiguous. When this happens you must guide the compiler by adding some extra information in the form of characters associated with the literal value. The following code shows these characters:

```
//:c04:Literals.cs
using System;

public class Literals {
  //!char c = 0xffff; // max char hex value
  byte b = 0x7f;      // max byte hex value
  short s = 0x7fff;   // max short hex value
  int i1 = 0x2f;      // Hexadecimal (lowercase)
  int i2 = 0X2F;      // Hexadecimal (uppercase)
  // Hex also works with long.
```

```
  long n1 = 200L;     // long suffix
  long n2 = 200l; // long suffix - generates warning
  long n3 = 200;
  //! long l6(200);   // not allowed
  float f1 = 1;
  float f2 = 1F;      // float suffix
  float f3 = 1f;      // float suffix
  float f4 = 1e-45f;  // 10 to the power
  float f5 = 1e+9f;   // float suffix
  double d1 = 1d;     // double suffix
  double d2 = 1D;     // double suffix
  double d3 = 47e47d; // 10 to the power
}///:~ (Non-executable code snippet)
```

Hexadecimal (base 16), which works with all the integral data types, is denoted by a leading **0x** or **0X** followed by 0—9 and a—f either in upper or lowercase. If you try to initialize a variable with a value bigger than it can hold (regardless of the numerical form of the value), the compiler will give you an error message. Notice in the above code the maximum possible hexadecimal values for **char**, **byte,** and **short**. If you exceed these, the compiler will automatically make the value an **int** and tell you that you need a narrowing cast for the assignment. You'll know you've stepped over the line.

A trailing character after a literal value establishes its type. Upper or lowercase **L** means **long**, upper or lowercase **F** means **float** and upper or lowercase **D** means **double**.

Exponents use a notation that some find rather dismaying: **1.39 e-47f**. In science and engineering, 'e' refers to the base of natural logarithms, approximately 2.718. (A more precise **double** value is available in C# as **Math.E**.) This is used in exponentiation expressions such as 1.39 x e^{-47}, which means 1.39 x 2.718^{-47}. However, when FORTRAN was invented they decided that **e** would naturally mean "ten to the power," which is an odd decision because FORTRAN was designed for science and engineering and one would think its designers would be sensitive about introducing such an ambiguity. At any rate, this custom was followed in C, C++ and now C#. So if you're used to thinking in terms of **e** as the base of natural logarithms, you must do a mental translation when you see an expression such as **1.39 e-47f** in C#; it means 1.39 x 10^{-47}.

Note that you don't need to use the trailing character when the compiler can figure out the appropriate type. With

```
long n3 = 200;
```

there's no ambiguity, so an **L** after the 200 would be superfluous. However, with

```
float f4 = 1e-47f; // 10 to the power
```

the compiler normally takes exponential numbers as doubles, so without the trailing **f** it will give you an error telling you that you must use a cast to convert **double** to **float**.

Promotion

You'll discover that if you perform any mathematical or bitwise operations on primitive data types that are smaller than an **int** (that is, **char**, **byte,** or **short**), those values will be promoted to **int** before performing the operations, and the resulting value will be of type **int**. So if you want to assign back into the smaller type, you must use a cast. (And, since you're assigning back into a smaller type, you might be losing information.) In general, the largest data type in an expression is the one that determines the size of the result of that expression; if you multiply a **float** and a **double**, the result will be **double**; if you add an **int** and a **long**, the result will be **long**.

C# has sizeof

The **sizeof( )** operator satisfies a specific need: it tells you the number of bytes allocated for data items. The most compelling need for **sizeof( )** in C and C++ is portability. Different data types might be different sizes on different machines, so the programmer must find out how big those types are when performing operations that are sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store integers as 16 bits. Programs could store larger values in integers on the first machine. As you might imagine, portability is a huge headache for C and C++ programmers. In C#, this most common use of sizeof() is not relevant, but it can come into play when interfacing with external data structures or when you're manipulating blocks of raw data and you're willing to forego convenience and safety for every last bit of speed (say, if you're writing a routine for processing video data). The **sizeof( )** operator is only usable inside **unsafe** code (see page 368).

C#'s preprocessor

C#'s preprocessing directives should be used with caution. Preprocessing is, as the name implies, something that happens before the human-readable code is transformed into the Common Intermediate Language that is the .NET equivalent of machine code. C# does not actually have a separate preprocessing step that runs prior to compilation but the form and use of these statements is intended to be familiar to C and C++ programmers.

While there's no harm in the **#region** directives, which are simply outlining clues to the Visual Studio .NET IDE, the other directives support *conditional compilation*, which allows a single codebase to generate multiple binary outputs. Preprocessing directives such as **#if**, **#else**, **#elif,** and **#endif** modify the code that is seen by the compiler. The variables that control which **#if...#else** block will be compiled may be modified on the compiler command-line, and therein lies the challenge. You essentially are defining different programs; at best, each preprocessor variant must be tested separately, at worst, incorrect behavior arises in some variants, a needed change is made only in one variant, etc.

The most common use of conditional compilation is to remove debugging behavior from a shipping product; this is done by defining a symbol on the compilation command-line, and using the **#if**, **#endif**, **#else**, and **#elif** directives to create conditional logic depending on the existence or absence of one or more such symbols. Here's a simple example:

```
//:c04:CondComp.cs
//Demonstrates conditional compilation

class CondComp{
  public static void Main(){
#if DEBUG
    Console.WriteLine("Debug behavior");
#endif
  }
}///:~
```

If **CondComp** is compiled with the command-line **csc /define:Debug CondComp.cs** it will print the line; if with a straight **csc CondComp.cs**, it won't. While this seems like a reasonable idea, in practice it often leads to situations where a change is made in one conditional branch and not in another, and the preprocessor leaves no trace in the code of the compilation options; in general, it's a better idea to use a **readonly bool** for such things. A reasonable compromise is to use the preprocessor directives to set the values of variables that are used to change runtime behavior:

```
//:c04:MarkedCondComp.cs
//Demonstrates conditional compilation
using System;

class CondComp {
  static readonly bool DEBUG =
#if DEBUG
```

```
  true;
#else
  false;
#endif

  public static void Main(){
    if (DEBUG)
      Console.WriteLine("Debug behavior");
  }
}///:~
```

In **MarkedCondComp**, if a problem arose, a debugger or logging facility would be able to read the value of the **DEBUG bool**, thus allowing the maintenance programmers to determine the compilation commands that lead to the troublesome behavior. The trivial disadvantages of this model are the slight penalty of a runtime comparison and the increase in the assembly's size due to the presence of the debugging code.

Precedence revisited

Operator precedence is difficult to remember, but here is a helpful mnemonic : "Ulcer Addicts Really Like C# A Lot."

Mnemonic	Operator type	Operators
Ulcer	Unary	+ - ++--
Addicts	Arithmetic (and shift)	* / % + - << >>
Really	Relational	> < >= <= == !=
Like	Logical (and bitwise)	&& || & | ^
C#	Conditional (ternary)	A > B ? X : Y
A Lot	Assignment	= (and compound assignment like *=)

Of course, with the shift and bitwise operators distributed around the table it is not a perfect mnemonic, but for non-bit operations it works.

A compendium of operators

The following example shows which primitive data types can be used with particular operators. Basically, it is the same example repeated over and over, but

using different primitive data types. The file will compile without error because the lines that would cause errors are commented out with a **//!**.

```
//:c04:AllOps.cs
namespace c03{
  using System;
// Tests all the operators on all the
// primitive data types to show which
// ones are accepted by the C# compiler.

  public class AllOps {
    // To accept the results of a boolean test:
    void F(bool b) {}
    void BoolTest(bool x, bool y) {
      // Arithmetic operators:
      //! x = x * y;
      //! x = x / y;
      //! x = x % y;
      //! x = x + y;
      //! x = x - y;
      //! x++;
      //! x--;
      //! x = +y;
      //! x = -y;
      // Relational and logical:
      //! F(x > y);
      //! F(x >= y);
      //! F(x < y);
      //! F(x <= y);
      F(x == y);
      F(x != y);
      F(!y);
      x = x && y;
      x = x || y;
      // Bitwise operators:
      //! x = ~y;
      x = x & y;
      x = x | y;
      x = x ^ y;
      //! x = x << 1;
      //! x = x >> 1;
```

```
    //! x = x >>> 1;
    // Compound assignment:
    //! x += y;
    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! char c = (char)x;
    //! byte B = (byte)x;
    //! short s = (short)x;
    //! int i = (int)x;
    //! long l = (long)x;
    //! float f = (float)x;
    //! double d = (double)x;
  }
  void CharTest(char x, char y) {
    // Arithmetic operators:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char)+y;
    x = (char)-y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
```

```
  //! F(x && y);
  //! F(x || y);
  // Bitwise operators:
  x= (char)~y;
  x = (char)(x & y);
  x  = (char)(x | y);
  x = (char)(x ^ y);
  x = (char)(x << 1);
  x = (char)(x >> 1);
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Casting:
  //! bool b = (bool)x;
  byte B = (byte)x;
  short s = (short)x;
  int i = (int)x;
  long l = (long)x;
  float f = (float)x;
  double d = (double)x;
}
void ByteTest(byte x, byte y) {
  // Arithmetic operators:
  x = (byte)(x* y);
  x = (byte)(x / y);
  x = (byte)(x % y);
  x = (byte)(x + y);
  x = (byte)(x - y);
  x++;
  x--;
  x = (byte)+ y;
  x = (byte)- y;
  // Relational and logical:
```

```
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void ShortTest(short x, short y) {
    // Arithmetic operators:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
```

```
x = (short)(x + y);
x = (short)(x - y);
x++;
x--;
x = (short)+y;
x = (short)-y;
// Relational and logical:
F(x > y);
F(x >= y);
F(x < y);
F(x <= y);
F(x == y);
F(x != y);
//! F(!x);
//! F(x && y);
//! F(x || y);
// Bitwise operators:
x = (short)~y;
x = (short)(x & y);
x = (short)(x | y);
x = (short)(x ^ y);
x = (short)(x << 1);
x = (short)(x >> 1);
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! bool b = (bool)x;
char c = (char)x;
byte B = (byte)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
```

```
  double d = (double)x;
}
void IntTest(int x, int y) {
  // Arithmetic operators:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relational and logical:
  F(x > y);
  F(x >= y);
  F(x < y);
  F(x <= y);
  F(x == y);
  F(x != y);
  //! F(!x);
  //! F(x && y);
  //! F(x || y);
  // Bitwise operators:
  x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  x = x << 1;
  x = x >> 1;
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x &= y;
  x ^= y;
  x |= y;
```

```
  // Casting:
  //! bool b = (bool)x;
  char c = (char)x;
  byte B = (byte)x;
  short s = (short)x;
  long l = (long)x;
  float f = (float)x;
  double d = (double)x;
}
void LongTest(long x, long y) {
  // Arithmetic operators:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relational and logical:
  F(x > y);
  F(x >= y);
  F(x < y);
  F(x <= y);
  F(x == y);
  F(x != y);
  //! F(!x);
  //! F(x && y);
  //! F(x || y);
  // Bitwise operators:
  x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  x = x << 1;
  x = x >> 1;
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
```

```
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
  }
  void FloatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
```

```
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
  }
  void DoubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
```

```
      F(x == y);
      F(x != y);
      //! F(!x);
      //! F(x && y);
      //! F(x || y);
      // Bitwise operators:
      //! x = ~y;
      //! x = x & y;
      //! x = x | y;
      //! x = x ^ y;
      //! x = x << 1;
      //! x = x >> 1;
      //! x = x >>> 1;
      // Compound assignment:
      x += y;
      x -= y;
      x *= y;
      x /= y;
      x %= y;
      //! x <<= 1;
      //! x >>= 1;
      //! x >>>= 1;
      //! x &= y;
      //! x ^= y;
      //! x |= y;
      // Casting:
      //! bool b = (bool)x;
      char c = (char)x;
      byte B = (byte)x;
      short s = (short)x;
      int i = (int)x;
      long l = (long)x;
      float f = (float)x;
    }
  }
}///:~ (non-executable code snippet)
```

Note that **bool** is quite limited. You can assign to it the values **true** and **false**, and you can test it for truth or falsehood, but you cannot add booleans or perform any other type of operation on them.

In **char**, **byte**, and **short** you can see the effect of promotion with the arithmetic operators. Each arithmetic operation on any of those types results in an **int** result, which must be explicitly cast back to the original type (a narrowing conversion that might lose information) to assign back to that type. With **int** values, however, you do not need to cast, because everything is already an **int**. Don't be lulled into thinking everything is safe, though. If you multiply two **int**s that are big enough, you'll overflow the result. The following example demonstrates this:

```
//:c04:OverFlow.cs
using System;
public class Overflow {
  public static void Main() {
    int big = 0x7fffffff; // max int value
    Console.WriteLine("big = " + big);
    int bigger = big * 4;
    Console.WriteLine("bigger = " + bigger);
    bigger = checked(big * 4);
    //! Console.WriteLine("never reached");
  }
}///:~
```

The output of this is:

```
big = 2147483647
bigger = -4

Unhandled Exception: System.OverflowException: Exception of
type System.OverflowException was thrown.
   at Overflow.Main()
```

If a potentially overflowing mathematical operation is not wrapped in the **checked( )** keyword, you will get no errors or warnings from the compiler, and no exceptions at run-time. (Exceptions have all of Chapter 11 devoted to them.)

Compound assignments do *not* require casts for **char**, **byte,** or **short**, even though they are performing promotions that have the same results as the direct arithmetic operations. On the other hand, the lack of the cast certainly simplifies the code.

You can see that, with the exception of **bool**, any primitive type can be cast to any other primitive type. Again, you must be aware of the effect of a narrowing conversion when casting to a smaller type, otherwise you might unknowingly lose information during the cast.

Execution control

C# uses all of C's execution control statements, so if you've programmed with C or C++ then most of what you see will be familiar. Most procedural programming languages have some kind of control statements, and there is often overlap among languages. In C#, the keywords include **if-else**, **while**, **do-while**, **for**, **foreach,** and a selection statement called **switch**. C# jumping keywords are **break, continue**, **goto** (yes, **goto**), and **return**.

true and false

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is **A == B**. This uses the conditional operator == to see if the value of **A** is equivalent to the value of **B**. The expression returns **true** or **false**. Any of the relational operators you've seen earlier in this chapter can be used to produce a conditional statement. Note that C# doesn't allow you to use a number as a **bool**, even though it's allowed in C and C++ (where truth is nonzero and falsehood is zero) and in Visual Basic (where truth is zero and falsehood non-zero). If you want to use a non-**bool** in a **bool** test, such as **if(a)**, you must first convert it to a **bool** value using a conditional expression, such as **if(a != 0)**.

if-else

The **if-else** statement is probably the most basic way to control program flow. The **else** is optional, so you can use **if** in two forms:

```
if(Boolean-expression)
  statement
```

or

```
if(Boolean-expression)
  statement
else
  statement
```

The conditional must produce a **bool** result. The *statement* means either a simple statement terminated by a semicolon or a compound statement, which is a group of simple statements enclosed in braces. Any time the word "*statement*" is used, it always implies that the statement can be simple or compound.

As an example of **if-else**, here is a **test( )** method that will tell you whether a guess is above, below, or equivalent to a target number:

```
//:c04:IfElse.cs
using System;

public class IfElse {
  static int Test(int testval, int target) {
    int result = 0;
    if (testval > target)
      result = +1;
    else if (testval < target)
      result = -1;
    else
      result = 0; // Match
    return result;
  }
  public static void Main() {
    Console.WriteLine(Test(10, 5));
    Console.WriteLine(Test(5, 10));
    Console.WriteLine(Test(5, 5));
  }
}///:~
```

It is conventional to indent the body of a control flow statement so the reader might easily determine where it begins and ends.

return

The **return** keyword has two purposes: it specifies what value a method will return (if it doesn't have a **void** return value) and it causes that value to be returned immediately. The **test( )** method above can be rewritten to take advantage of this:

```
//:c04:IfElse2.cs
using System;

public class IfElse2 {
  static int Test(int testval, int target) {
    if (testval > target)
      return 1;
    else if (testval < target)
      return -1;
    else
      return 0; // Match
  }
```

```
  public static void Main() {
    Console.WriteLine(Test(10, 5));
    Console.WriteLine(Test(5, 10));
    Console.WriteLine(Test(5, 5));
  }
}///:~
```

Although this code has two **else**'s, they are actually unnecessary, because the method will not continue after executing a **return**. It is good programming practice to have as few exit points as possible in a method; a reader should be able to see at a glance the "shape" of a method without having to think "Oh! Unless something happens in this conditional, in which case it never gets to this other area." After rewriting the method so that there's only one exit point, we can add extra functionality to the method and know that it will always be called:

```
//:c04:IfElse3.cs
using System;

public class IfElse3 {
  static int Test(int testval, int target) {
    int result = 0;  //Match
    if (testval > target)
      result = 1;
    else if (testval < target)
      result = -1;
    Console.WriteLine("All paths pass here");
    return result;
  }
  public static void Main() {
    Console.WriteLine(Test(10, 5));
    Console.WriteLine(Test(5, 10));
    Console.WriteLine(Test(5, 5));
  }
} ///:~
```

Iteration

while, **do-while,** and **for** control looping and are sometimes classified as *iteration statements*. A *statement* repeats until the controlling *Boolean-expression* evaluates to false. The form for a **while** loop is

```
while(Boolean-expression)
  statement
```

The *Boolean-expression* is evaluated once at the beginning of the loop and again before each further iteration of the *statement*.

Here's a simple example that generates random numbers until a particular condition is met:

```
//:c04:WhileTest.cs
using System;
// Demonstrates the while loop.
public class WhileTest {
  public static void Main() {
    Random rand = new Random();
    double r = 0;
    while (r < 0.99d) {
      r = rand.NextDouble();
      Console.WriteLine(r);
    }
  }
}///:~
```

This uses the **static** method **NextDouble( )** in the **Random** class, which generates a **double** value between 0 and 1. (It includes 0, but not 1.) The conditional expression for the **while** says "keep doing this loop until the number is 0.99 or greater." Each time you run this program you'll get a different-sized list of numbers.

do-while

The form for **do-while** is

```
do
  statement
while(Boolean-expression);
```

The sole difference between **while** and **do-while** is that the statement of the **do-while** always executes at least once, even if the expression evaluates to false the first time. In a **while**, if the conditional is false the first time the statement never executes. In practice, **do-while** is less common than **while**.

for

A **for** loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of "stepping." The form of the **for** loop is:

```
for(initialization; Boolean-expression; step)
  statement
```

Any of the expressions *initialization*, *Boolean-expression* or *step* can be empty. The expression is tested before each iteration, and as soon as it evaluates to **false** execution will continue at the line following the **for** statement. At the end of each loop, the *step* executes.

for loops are usually used for "counting" tasks:

```
//:c04:ListCharacters.cs
using System;
// Demonstrates "for" loop by listing
// all the ASCII characters.
public class ListCharacters {
  public static void Main() {
    for ( char c = (char) 0; c < (char) 128; c++)
      if (c != 26 )  // ANSI Clear screen
        Console.WriteLine("value: " + (int)c +
          " character: " + c);
  }
}///:~
```

Note that the variable **c** is defined at the point where it is used, inside the control expression of the **for** loop, rather than at the beginning of the block denoted by the open curly brace. The scope of **c** is the expression controlled by the **for**.

Traditional procedural languages like C require that all variables be defined at the beginning of a block so when the compiler creates a block it can allocate space for those variables. In C#, you can spread your variable declarations throughout the block, defining them at the point that you need them. This allows a more natural coding style and makes code easier to understand.

You can define multiple variables within a **for** statement, but they must be of the same type:

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
  /* body of for loop */;
```

The **int** definition in the **for** statement covers both **i** and **j**. The ability to define variables in the control expression is limited to the **for** loop. You cannot use this approach with any of the other selection or iteration statements.

foreach

C# has a specialized iteration operator called **foreach**. Unlike the others, **foreach** does not loop based on a boolean expression. Rather, it executes a block of code on each element in an array or other collection. The form for **foreach** is:

```
foreach(type loopVariable in collection){
   statement
}
```

The **foreach** statement is a terse way to specify the most common type of loop and does so without introducing potentially confusing index variables. Compare the clarity of **foreach** and **for** in this example:

```
//:c04:ForEach.cs
using System;

class ForEach {
  public static void Main() {
    string[] months = {"January", "February",
      "March", "April"}; //etc
    string[] weeks = {"1st", "2nd", "3rd", "4th"};
    string[] days = {"Sunday", "Monday",
      "Tuesday", "Wednesday"}; //etc

    foreach(string month in months)
      foreach(string week in weeks)
        foreach(string day in days)
          Console.WriteLine("{0} {1} week {2}",
                            month, week, day);
    for (int month = 0;
        month < months.Length;
        month++) {
      for (int week = 0;
          week < weeks.Length;
          week++) {
        for (int day = 0;
            day < days.Length;
            day++) {
          Console.WriteLine(
            "{0} {1} week {2}",
            months[month], weeks[week],days[day]);
        }
```

```
      }
    }
  }
} ///:~
```

Another advantage of **foreach** is that it performs an implicit typecast on objects stored in collections, saving a few more keystrokes when objects are stored not in arrays, but in more complex data structures. We'll cover this aspect of **foreach** in Chapter 10.

The comma operator

Earlier in this chapter we stated that the comma *operator* (not the comma *separator*, which is used to separate definitions and function arguments) has only one use in C#: in the control expression of a **for** loop. In both the initialization and step portions of the control expression you can have a number of statements separated by commas, and those statements will be evaluated sequentially. The previous bit of code uses this ability. Here's another example:

```
//:c04:ListCharacters2.cs
using System;
public class CommaOperator {
  public static void Main() {
    for (int i = 1, j = i + 10; i < 5;
        i++, j = i * 2) {
      Console.WriteLine("i= " + i + " j= " + j);
    }
  }
}///:~
```

Here's the output:

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

You can see that in both the initialization and step portions the statements are evaluated in sequential order. Also, the initialization portion can have any number of definitions *of one type*.

break **and** continue

Inside the body of any of the iteration statements you can also control the flow of the loop by using **break** and **continue**. **break** quits the loop without executing

the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

This program shows examples of **break** and **continue** within **for** and **while** loops:

```
//:c04:BreakAndContinue.cs
// Demonstrates break and continue keywords.
using System;

public class BreakAndContinue {
  public static void Main() {
    int i = 0;
    for (i = 0; i < 100; i++) {
      if (i == 74) break; // Out of for loop
      if (i % 9 != 0) continue; // Next iteration
      Console.WriteLine(i);
    }
    i = 0;
    // An "infinite loop":
    while (true) {
      i++;
      int j = i * 27;
      if (j == 1269) break; // Out of loop
      if (i % 10 != 0) continue; // Top of loop
      Console.WriteLine(i);
    }
  }
}///:~
```

In the **for** loop the value of **i** never gets to 100 because the **break** statement breaks out of the loop when **i** is 74. Normally, you'd use a **break** like this only if you didn't know when the terminating condition was going to occur. The **continue** statement causes execution to go back to the top of the iteration loop (thus incrementing **i**) whenever **i** is not evenly divisible by 9. When it is, the value is printed.

The second portion shows an "infinite loop" that would, in theory, continue forever. However, inside the loop there is a **break** statement that will break out of the loop. In addition, you'll see that the **continue** moves back to the top of the loop without completing the remainder. (Thus printing happens in the second loop only when the value of **i** is divisible by 10.) The output is:

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

The value 0 is printed because 0 % 9 produces 0.

A second form of the infinite loop is **for(;;)**. The compiler treats both **while(true)** and **for(;;)** in the same way, so whichever one you use is a matter of programming taste. (Often, people from C backgrounds think **for(;;)** is clearer, although **while(true)** certainly seems more universal.)

The infamous goto

The **goto** keyword has been present in programming languages from the beginning. Indeed, **goto** was the genesis of program control in assembly language: "if condition A, then jump here, otherwise jump there." If you read the assembly code that is ultimately generated by virtually any compiler, you'll see that program control contains many jumps. However, a **goto** is a jump at the source-code level, and that's what brought it into disrepute. If a program will always jump from one point to another, isn't there some way to reorganize the code so the flow of control is not so jumpy? **goto** fell into true disfavor with the publication of the famous 1968 letter "Go To Statement Considered Harmful" by Edsger Dijkstra (http://www.acm.org/classics/oct95/). Dijkstra (who passed away shortly before this book went to press) argued that when you have a jump, the context that created the program state becomes difficult to visualize. Since then, goto-bashing has been a popular sport, with advocates of the cast-out keyword scurrying for cover.

As is typical in situations like this, the middle ground is the most fruitful. The problem is not the use of **goto**, but the overuse of **goto** or, indeed, any statement that makes it difficult to say "When this line is reached, the state of the system is necessarily such-and-such." The best way to write code that makes system state easy to determine is to minimize cyclomatic complexity, which is a fancy way of

saying “use as few selection and jump statements as possible.” Cyclomatic complexity is the measure of the number of possible paths through a block of code.

```
class Cyclomatic{
    public void simple(){
        int x = 0;
        int y = x;
        int z = y + y;
        System.Console.WriteLine(z);
    }

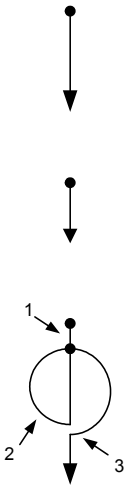
    public int alsoSimple(int x){
        int y = x;
        return y + y;
    }

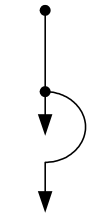
    public void oneLoop(int x){
        System.Console.WriteLine("Begin");
        for(int y = x; y < 10; y++){
            int z = x + y;
            System.Console.WriteLine(z);
        }
        System.Console.WriteLine("Done");
    }

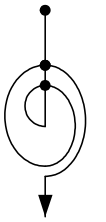
    public void twoExits(){
        Random r = new Random();
        int x = r.Next();
        if(x % 2 == 0){
            System.Console.WriteLine("Even");
            return;
        }
        System.Console.WriteLine("Odd");
    }

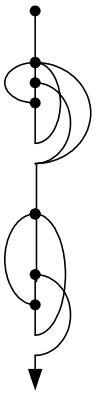
    public void twoLoop(){
        int x = 1;
        for(int y = x; y < 10; y++){
            for(int z = x + y; z < 6; z++){
                System.Console.WriteLine(z);
            }
        }
        System.Console.WriteLine("Done");
    }

    public void spaghetti() {
    int i = 0;
    for(i = 0; i < 100; i++) {
      if(i == 74) break;
      if(i % 9 != 0) continue;
      System.Console.WriteLine(i);
    }
    i = 0;
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break;
      if(i % 10 != 0) continue;
      System.Console.WriteLine(i);
    }
  }
}
```









In the figure on the previous page, the methods **simple( )** and **alsoSimple( )** have a cyclomatic complexity of 1; there is only a single path through the code. It does not matter how long the method is, whether the method creates objects, or even if the method calls other, more complex, methods (if those methods have high complexity, so be it; it doesn't affect the complexity of the method at hand). This simplicity is reflected in the control graph shown: a single line showing the direction of execution towards the exit point.

The method **oneLoop( )** is slightly more complex. No matter what its input parameter, it will print out "Begin" and assign x to y at the very beginning of the for loop. That's the first edge on its control graph (to help align with the source code, the figure shows a single "edge" as a straight length of code and a curving jump). Then, it *may* continue into the loop, assign z and print it, increment y, and loop; that's the second edge. Finally, at some point, y will be equal to 10 and control will jump to the end of the method. This is the third edge, as marked on the figure. Method twoExits() also has a cyclomatic complexity of 3, although its second edge doesn't loop, but exits.

The next method, twoLoops(), hardly seems more complex than oneLoop(), but if you look at its control graph, you can count five distinct edges. Finally, we see a visual representation of what programmers call "spaghetti code." With a cyclomatic complexity of 12, spaghetti() is about as complex as a method should *ever* be. Once a method has more than about six conditional and iteration operators, it starts to become difficult to understand the ramifications of any changes. In the 1950s, the psychologist George Miller published a paper that said that "Seven plus or minus two" is the limit of our "span of absolute judgment." Trying to keep more than this number of things in our head at once is very error-prone. Luckily, we have this thing called "writing" (or, in our case, coding C#) which allows us to break the problem of "absolute judgment" into successive sub-problems, which can then be treated as units for the purpose of making higher-level judgments. Sounds like computer programming to me!

(The paper points out that by increasing the dimension of visual variables, we can achieve astonishing levels of discrimination as we do, say, when we recognize a friend we've not seen in years while rushing through an airport. It's interesting to note that computer programming hardly leverages this capacity at all. You can read the paper, which anticipates exactly the sort of thinking and choice-making common to programming, at http://www.well.com/user/smalin/miller.html.)

In C#, **goto** can be used to jump within a method to a label. A label is an identifier followed by a colon, like this:

```
label1:
```

Although it's legal to place a label anywhere in a method, the only place where it's a good idea is right before an iteration statement. And the sole reason to put a label before an iteration is if you're going to nest another iteration or a switch inside it. That's because while **break** and **continue** interrupt only the loop that contains them, **goto** can interrupt the loops up to where the label exists. Here is an example of the use and abuse of **goto**:

```
//:c04:Goto.cs
// Using Goto
using System;

public class Goto {
  public static void Main() {
    int i = 0;
    Random rand = new Random();
    outer: //Label before iterator
    for (; true ;) { // infinite loop
      Console.WriteLine("Prior to inner loop");
      inner: // Another label
      for (; i < 10; i++) {
        Console.WriteLine("i = " + i);
        if (i == 7) {
          Console.WriteLine("goto outer");
          i++; // Otherwise i never
               // gets incremented.
          goto outer;
        }
        if (i == 8) {
          Console.WriteLine("goto inner");
          i++; //Otherwise i never
          //gets incremented
          goto inner;
        }
        double d = rand.NextDouble();
        if (i == 9 && d < 0.6) {
          Console.WriteLine("Legal but terrible");
          goto badIdea;
        }
        Console.WriteLine("Back in the loop");
        if (i == 9)
          goto bustOut;
```

```
      }
    }
    bustOut:
    Console.WriteLine("Exit loop");
    if (rand.NextDouble() < 0.5) {
      goto spaghettiJump;
    }
    badIdea:
    Console.WriteLine("How did I get here?");
    goto outer;
    spaghettiJump:
    Console.WriteLine("Don't ever, ever do this.");
  }
} ///:~
```

Things start out appropriately enough, with the labeling of the two loops as **outer** and **inner**. After counting to 7 and getting lulled into a false sense of security, control jumps out of both loops, and re-enters following the **outer** label. On the next loop, control jumps to the **inner** label. Then things get weird: if the random number generator comes up with a value less than 0.6, control jumps downwards, to the label marked **badIdea**, the method prints "How did I get here?" and then jumps all the way back to the **outer** label. On the next run through the inner loop, i is still equal to 9 but, eventually, the random number generator will come up with a value that will skip the jump to **badIdea** and print that we're "back in the loop." Then, instead of using the **for** statement's terminating condition, we decide that we're going to jump to the **bustOut** label. We do the programmatic equivalent of flipping a coin and either "fall through" into the **badIdea** area (which, of course, jumps us back to **outer**) or jump to the **spaghettiJump** label.

So why is this code considered so terrible? For one thing, it has a high cyclomatic complexity – it's just plain confusing. Also, note how much harder it is to understand program flow when one can't rely on brackets and indenting. And to make things worse, let's say you were debugging this code and you placed a breakpoint at the line `Console.WriteLine("How did I get here?")`. When the breakpoint is reached, there is no way, short of examining the output, for you to determine whether you reached it from the jump from the inner loop or from falling through from the immediately preceding lines (in this case, the program's output is sufficient to this cause, but in the real world of complex systems, GUIs, and Web Services, it never is). As Dijkstra put it, "it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress."

By "coordinates" Dijkstra meant a way to know the path by which a system arrived in its current state. It's only with such a path in hand that one can debug, since challenging defects only become apparent sometime *after* the mistake has been made. (It is, of course, common to make mistakes immediately or just before the problem becomes apparent, but such mistakes aren't hard to root out and correct.) Dijkstra went on to say that his criticism was not just about **goto**, but that all language constructs "*should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.*" We'll revisit this concern when speaking of the way that C# and the .NET framework handle exceptions (obeying the requirement) and threading (which doesn't).

switch

The **switch** is sometimes classified as a *selection statement*. The **switch** statement selects from among pieces of code based on the value of an integral expression. Its form is:

```
switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; return;
  case integral-value3 : statement; continue;
  case integral-value4 : statement; throw exception;
  case integral-value5 : statement;
                         goto external-label;
  case integral-value6 : //No statements
  case integral-value7 : statement;
                         goto case integral-value;
  // ...
  default: statement; break;
}
```

Integral-selector is an expression that produces an integral value. The **switch** compares the result of *integral-selector* to each *integral-value*. If it finds a match, the corresponding *statement* (simple or compound) executes. If no match occurs, the **default** *statement* executes.

You will notice in the above definition that each **case** ends with some kind of jump statement. The first one shown, **break**, is by far the most commonly used. Note that **goto** can be used in both the form discussed previously, which jumps to an arbitrary label in the enclosing statement block, and in a new form, **goto case**, which transfers control to the specified case block.

Unlike Java and C++, each case block, including the default block, must end in a jump statement. There is no "fall-through," although if a selector contains no statements at all, it may immediately precede another selector. In the definition, this is seen at the selector for integral-value6, which will execute the statements in integral-value7's case block.

The **switch** statement is a clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to a predefined type such as **int**, **char**, or **string**, or to an enumeration. For other types, you must use either a series of **if** statements, or implement some kind of conversion to one of the supported types. More generally, a well-designed object-oriented program will generally move a lot of control switching away from explicit tests in code into polymorphism (which we'll get to in Chapter 8).

Here's an example that creates letters randomly and determines whether they're vowels or consonants:

```
//:c04:VowelsAndConsonants.cs
//Demonstrates the switch statement.
using System;
public class VowelsAndConsonants {
  public static void Main() {
    Random rand = new Random();
    for (int i = 0; i < 100; i++) {
      char c = (char)(rand.Next('a','z' + 1));
      Console.WriteLine(c + ": ");
      switch (c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
        Console.WriteLine("vowel");
        break;
      case 'y':
        Console.WriteLine("Sometimes a vowel");
        break;
      default:
        Console.WriteLine("consonant");
        break;
      }
```

```
      }
    }
}///:~
```

Since **char**s can be implicitly promoted to **int**s, Random.Next(int lowerBound, int upperBound) can be used to return values in the appropriate ASCII range.

Summary

This chapter concludes the study of fundamental features that appear in most programming languages: calculation, operator precedence, type casting, and selection and iteration. Now you're ready to begin taking steps that move you closer to the world of object-oriented programming. The next chapter will cover the important issues of initialization and cleanup of objects, followed in the subsequent chapter by the essential concept of implementation hiding.

Exercises

1. There are two expressions in the section labeled "precedence" early in this chapter. Put these expressions into a program and demonstrate that they produce different results.
2. Put the methods **Ternary( )** and **Alternative( )** into a working program.
3. Write a program that prints values from one to 100.
4. Modify Exercise 3 so that the program exits by using the **break** keyword at value 47. Try using **return** instead.
5. Write a function that takes two string arguments, and uses all the bool comparisons to compare the two strings and print the results. In **Main( )**, call your function with some different string objects.
6. Write a program that generates 25 random int values. For each value, use an **if-else** statement to classify it as greater than, less than or equal to a second randomly-generated value.
7. Modify Exercise 6 so that your code is surrounded by an "infinite" **while** loop. It will then run until you interrupt it from the keyboard (typically by pressing Control-C).

8. Write a program that uses two nested for loops and the modulus operator (%) to detect and print prime numbers (integral numbers that are not evenly divisible by any other numbers except for themselves and 1).

9. Modify the solution to Exercise 8 so that it uses a **foreach** statement to test every integer between 2 and 10000 for primality.

10. Create a **switch** statement that prints a message for each case, and put the **switch** inside a **for** loop that tries each case. Put a **break** after each case and test it, then remove the **break**s and see what happens.

11. Referring back to Exercises 15-17, write a program that "performs" the complex behavior from exercise 1-17 by writing to the Console a description of the behavior and the class doing it. *Modify your previous classes as necessary* to accommodate the task.

12. On a large piece of paper or whiteboard, draw a box with the name of each class used in exercise 11 (one will be **Console**). One class will contain the **Main( )** method that is the entry point to your program. Place a coin on that class. Go through the program you wrote for Exercise 11 line-by-line, tracing the execution of your program by moving the coin into the class that is responsible for that line. As you "visit" a class, write the name of the method called or property accessed in the box. The coin should "visit" every class that is collaborating to accomplish the task.

13. Repeat Exercises 1-17, 11, and 12 (describe a complex behavior, implement it, and trace execution on a diagram). Choose a behavior that uses at least two of the classes used in the first go-round. Are some classes being burdened with all the work, while other classes turn out to be unnecessary? If so, can you see a way to restructure the classes so that the work is more evenly distributed? Are there any methods that are used in both solutions that have several lines of code in common? If so, eliminate this common code by refactoring it into a **private** method. Confirm that the program you wrote for Exercise 11 continues to work!

5: Initialization and Cleanup

An object-oriented solution consists of a "web" of connected objects describing the problem and a route to a solution. Like database programming, object-oriented programming involves the creation of a system structure that, although necessarily a digital will-o'-wisp, seems very tangible. As more and more systems have been built over the years, it has turned out that two of the most error-prone tasks are the *initialization* and *cleanup* of the objects that make up the system structure.

Many C bugs occur when the programmer forgets to initialize a variable. This is especially true with libraries when users don't know how to initialize a library component, or even that they must. Cleanup is a special problem because it's easy to forget about an element when you're done with it, since it no longer concerns you. Thus, the resources used by that element are retained and you can easily end up running out of resources (most notably, memory).

C++ introduced the concept of a *constructor* and a *destructor*, special methods automatically called when an object is created and destroyed. C# has these facilities, and in addition has a garbage collector that automatically releases memory resources when they're no longer being used. This chapter examines the issues of initialization and cleanup, and their support in C#.

Guaranteed initialization with the onstructor

You can imagine creating a method called **Initialize( )** for every class you write. The name is a hint that it should be called before using the object. Unfortunately, this means the user must remember to call the method. In C#, the class designer can guarantee initialization of every object by providing a special method called a *constructor*. If a class has a constructor, C# automatically calls that constructor

when an object is created, before users can even get their hands on it. So initialization is guaranteed.

The next challenge is what to name this method. There are two issues. The first is that any name you use could clash with a name you might like to use as a member in the class. The second is that because the compiler is responsible for calling the constructor, it must always know which method to call. The C++ solution seems the easiest and most logical, so it's also used in C#: the name of the constructor is the same as the name of the class. It makes sense that such a method will be called automatically on initialization.

Here's a simple class with a constructor:

```
//:c05:SimpleConstructor.cs
using System;

// Demonstration of a simple constructor.
public class Rock {
  public Rock() { // This is the constructor
    Console.WriteLine("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void Main() {
    for (int i = 0; i < 10; i++)
      new Rock();
  }
}///:~
```

Now, when an object is created:

```
new Rock();
```

storage is allocated and the constructor is called. It is guaranteed that the object will be properly initialized before you can get your hands on it.

Note that the name of the constructor must match the name of the class *exactly*.

Like any method, the constructor can have arguments to allow you to specify *how* an object is created. The above example can easily be changed so the constructor takes an argument:

```
//:c05:SimpleConstructor2.cs
```

```
using System;

// Demonstration of a simple constructor.
public class Rock2 {
  public Rock2(int i) { // This is the constructor
    Console.WriteLine("Creating Rock number: " + i);
  }
}

public class SimpleConstructor {
  public static void Main() {
    for (int i = 0; i < 10; i++)
      new Rock2(i);
  }
}///:~
```

Constructor arguments provide you with a way to provide parameters for the initialization of an object. For example, if the class **Tree** has a constructor that takes a single integer argument denoting the height of the tree, you would create a **Tree** object like this:

```
Tree t = new Tree(12);  // 12-foot tree
```

If **Tree(int)** is your only constructor, then the compiler won't let you create a **Tree** object any other way.

Constructors eliminate a large class of problems and make the code easier to read. In the preceding code fragment, for example, you don't see an explicit call to some **initialize( )** method that is conceptually separate from definition. In C#, definition and initialization are unified concepts—you can't have one without the other.

The constructor is an unusual type of method because it has no return value. This is distinctly different from a **void** return value, in which the method is declared explicity as returning nothing. With constructors you are not given a choice of what you return; a constructor always returns an object of the constructor's type. If there was a declared return value, and if you could select your own, the compiler would somehow need to know what to do with that return value. Accidentally typing a return type such as **void** before declaring a constructor is a common thing to do on a Monday morning, but the C# compiler won't allow it, telling you "member names cannot be the same as their enclosing type."

Method overloading

One of the important features in any programming language is the use of names. When you create an object, you give a name to a region of storage. A method is a name for an action. By using names to describe your system, you create a program that is easier for people to understand and change. It's a lot like writing prose—the goal is to communicate with your readers.

You refer to all objects and methods by using names. Well-chosen names make it easier for you and others to understand your code.

A problem arises when mapping the concept of nuance in human language onto a programming language. Often, the same word expresses a number of different meanings—it's *overloaded*. This is useful, especially when it comes to trivial differences. You say "wash the shirt," "wash the car," and "wash the dog." It would be silly to be forced to say, "shirtWash the shirt," "carWash the car," and "dogWash the dog" just so the listener doesn't need to make any distinction about the action performed. Most human languages are redundant, so even if you miss a few words, you can still determine the meaning. We don't need unique identifiers—we can deduce meaning from context.

Most programming languages (C in particular) require you to have a unique identifier for each function. So you could not have one function called **print( )** for printing integers and another called **print( )** for printing floats—each function requires a unique name.

In C# and other languages in the C++ family, another factor forces the overloading of method names: the constructor. Because the constructor's name is predetermined by the name of the class, there can be only one constructor name. But what if you want to create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way or by reading information from a file. You need two constructors, one that takes no arguments (the *default* constructor, also called the *no-arg* constructor), and one that takes a **string** as an argument, which is the name of the file from which to initialize the object. Both are constructors, so they must have the same name—the name of the class. Thus, *method overloading* is essential to allow the same method name to be used with different argument types. And although method overloading is a must for constructors, it's a general convenience and can be used with any method.

Here's an example that shows both overloaded constructors and overloaded ordinary methods:

```
//:c05:OverLoading.cs
// Demonstration of both constructor
// and ordinary method overloading.
using System;

class Tree {
  int height;
  public Tree() {
    Prt("Planting a seedling");
    height = 0;
  }
  public Tree(int i) {
    Prt("Creating new Tree that is "
        + i + " feet tall");
    height = i;
  }
  internal void Info() {
    Prt("Tree is " + height
        + " feet tall");
  }
  internal void Info(string s) {
    Prt(s + ": Tree is "
        + height + " feet tall");
  }
  static void Prt(string s) {
    Console.WriteLine(s);
  }
}

public class Overloading {
  public static void Main() {
    for (int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.Info();
      t.Info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
} ///:~
```

A **Tree** object can be created either as a seedling, with no argument, or as a plant grown in a nursery, with an existing height. To support this, there are two constructors, one that takes no arguments and one that takes the existing height.

You might also want to call the **info( )** method in more than one way: for example, with a **string** argument if you have an extra message you want printed, and without if you have nothing more to say. It would seem strange to give two separate names to what is obviously the same concept. Fortunately, method overloading allows you to use the same name for both.

Distinguishing overloaded methods

If the methods have the same name, how can C# know which method you mean? There's a simple rule: each overloaded method must take a unique list of argument types.

If you think about this for a second, it makes sense: how else could a programmer tell the difference between two methods that have the same name, other than by the types of their arguments?

Even differences in the ordering of arguments are sufficient to distinguish two methods although you don't normally want to take this approach, as it produces difficult-to-maintain code:

```
//:c05:OverLoadingOrder.cs
// Overloading based on the order of
// the arguments.
using System;

public class OverloadingOrder {
  static void Print(string s, int i) {
    Console.WriteLine(
      "string: " + s + ", int: " + i);
  }
  static void Print(int i, string s) {
    Console.WriteLine(
      "int: " + i + ", string: " + s);
  }
  public static void Main() {
    Print("string first", 11);
    Print(99, "Int first");
  }
} ///:~
```

The two **Print( )** methods have identical arguments, but the order is different, and that's what makes them distinct.

Overloading with primitives

A primitive can be automatically promoted from a smaller type to a larger one and this can be slightly confusing in combination with overloading. The following example demonstrates what happens when a primitive is handed to an overloaded method:

```
//:c05:PrimitiveOverloading.cs
// Promotion of primitives and overloading.
using System;

public class PrimitiveOverloading {
  // boolean can't be automatically converted
  static void Prt(string s) {
    Console.WriteLine(s);
  }

  void F1(char x) { Prt("F1(char)");}
  void F1(byte x) { Prt("F1(byte)");}
  void F1(short x) { Prt("F1(short)");}
  void F1(int x) { Prt("F1(int)");}
  void F1(long x) { Prt("F1(long)");}
  void F1(float x) { Prt("F1(float)");}
  void F1(double x) { Prt("F1(double)");}

  void F2(byte x) { Prt("F2(byte)");}
  void F2(short x) { Prt("F2(short)");}
  void F2(int x) { Prt("F2(int)");}
  void F2(long x) { Prt("F2(long)");}
  void F2(float x) { Prt("F2(float)");}
  void F2(double x) { Prt("F2(double)");}

  void F3(short x) { Prt("F3(short)");}
  void F3(int x) { Prt("F3(int)");}
  void F3(long x) { Prt("F3(long)");}
  void F3(float x) { Prt("F3(float)");}
  void F3(double x) { Prt("F3(double)");}

  void F4(int x) { Prt("F4(int)");}
```

```
void F4(long x) { Prt("F4(long)");}
void F4(float x) { Prt("F4(float)");}
void F4(double x) { Prt("F4(double)");}

void F5(long x) { Prt("F5(long)");}
void F5(float x) { Prt("F5(float)");}
void F5(double x) { Prt("F5(double)");}

void F6(float x) { Prt("F6(float)");}
void F6(double x) { Prt("F6(double)");}

void F7(double x) { Prt("F7(double)");}

void TestConstVal() {
  Prt("Testing with 5");
  F1(5);F2(5);F3(5);F4(5);F5(5);F6(5);F7(5);
}
void TestChar() {
  char x = 'x';
  Prt("char argument:");
  F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
}
void TestByte() {
  byte x = 0;
  Prt("byte argument:");
  F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
}
void TestShort() {
  short x = 0;
  Prt("short argument:");
  F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
}
void TestInt() {
  int x = 0;
  Prt("int argument:");
  F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
}
void TestLong() {
  long x = 0;
  Prt("long argument:");
  F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
```

```
  }
  void TestFloat() {
    float x = 0;
    Prt("Float argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  void TestDouble() {
    double x = 0;
    Prt("double argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  public static void Main() {
    PrimitiveOverloading p =
    new PrimitiveOverloading();
    p.TestConstVal();
    p.TestChar();
    p.TestByte();
    p.TestShort();
    p.TestInt();
    p.TestLong();
    p.TestFloat();
    p.TestDouble();
  }
} ///:~
```

If you view the output of this program, you'll see that the constant value 5 is treated as an **int**, so if an overloaded method is available that takes an **int** it is used. In all other cases, if you have a data type that is smaller than the argument in the method, that data type is promoted. **char** produces a slightly different effect, since if it doesn't find an exact **char** match, it is promoted to **int**.

What happens if your argument is *bigger* than the argument expected by the overloaded method? A modification of the above program gives the answer:

```
//:c05:Demotion.cs
// Demotion of primitives and overloading.
using System;

public class Demotion {
  static void Prt(string s) {
    Console.WriteLine(s);
  }
```

```
void F1(char x) { Prt("F1(char)");}
void F1(byte x) { Prt("F1(byte)");}
void F1(short x) { Prt("F1(short)");}
void F1(int x) { Prt("F1(int)");}
void F1(long x) { Prt("F1(long)");}
void F1(float x) { Prt("F1(float)");}
void F1(double x) { Prt("F1(double)");}

void F2(char x) { Prt("F2(char)");}
void F2(byte x) { Prt("F2(byte)");}
void F2(short x) { Prt("F2(short)");}
void F2(int x) { Prt("F2(int)");}
void F2(long x) { Prt("F2(long)");}
void F2(float x) { Prt("F2(float)");}

void F3(char x) { Prt("F3(char)");}
void F3(byte x) { Prt("F3(byte)");}
void F3(short x) { Prt("F3(short)");}
void F3(int x) { Prt("F3(int)");}
void F3(long x) { Prt("F3(long)");}

void F4(char x) { Prt("F4(char)");}
void F4(byte x) { Prt("F4(byte)");}
void F4(short x) { Prt("F4(short)");}
void F4(int x) { Prt("F4(int)");}

void F5(char x) { Prt("F5(char)");}
void F5(byte x) { Prt("F5(byte)");}
void F5(short x) { Prt("F5(short)");}

void F6(char x) { Prt("F6(char)");}
void F6(byte x) { Prt("F6(byte)");}

void F7(char x) { Prt("F7(char)");}

void TestDouble() {
  double x = 0;
  Prt("double argument:");
  F1(x);F2((float)x);F3((long)x);F4((int)x);
  F5((short)x);F6((byte)x);F7((char)x);
```

```
  }
  public static void Main() {
    Demotion p = new Demotion();
    p.TestDouble();
  }
} ///:~
```

Here, the methods take narrower primitive values. If your argument is wider then you must *cast* to the necessary type using the type name in parentheses. If you don't do this, the compiler will issue an error message.

You should be aware that this is a *narrowing conversion,* which means you might lose information during the cast. This is why the compiler forces you to do it—to flag the narrowing conversion.

Overloading on return values

It is common to wonder "Why only class names and method argument lists? Why not distinguish between methods based on their return values?" For example, these two methods, which have the same name and arguments, are easily distinguished from each other:

```
void f() {}
int f() {}
```

This works fine when the compiler can unequivocally determine the meaning from the context, as in **int x = f( )**. However, you can call a method and ignore the return value; this is often referred to as *calling a method for its side effect* since you don't care about the return value but instead want the other effects of the method call. So if you call the method this way:

```
f();
```

how can C# determine which **f( )** should be called? And how could someone reading the code see it? Because of this sort of problem, you cannot use return value types to distinguish overloaded methods.

Default constructors

As mentioned previously, a default constructor (a.k.a. a "no-arg" constructor) is one without arguments, used to create a "vanilla object." If you create a class that has no constructors, the compiler will automatically create a default constructor for you. For example:

```
//:c05:DefaultConstructor.cs
class Bird {
```

```
  int i;
}

public class DefaultConstructor {
  public static void Main() {
    Bird nc = new Bird(); // default!
  }
}///:~
```

The line

```
new Bird();
```

creates a new object and calls the default constructor, even though one was not explicitly defined. Without it we would have no method to call to build our object. However, if you define any constructors (with or without arguments), the compiler will *not* synthesize one for you:

```
class Bush {
  Bush(int i) {}
  Bush(double d) {}
}
```

Now if you say:

```
new Bush();
```

the compiler will complain that it cannot find a constructor that matches. It's as if when you don't put in any constructors, the compiler says "You are bound to need *some* constructor, so let me make one for you." But if you write a constructor, the compiler says "You've written a constructor so you know what you're doing; if you didn't put in a default it's because you meant to leave it out."

The this keyword

If you have two objects of the same type called **a** and **b**, you might wonder how it is that you can call a method **f( )** for both those objects:

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

If there's only one method called **f( )**, how can that method know whether it's being called for the object **a** or **b**?

To allow you to write the code in a convenient object-oriented syntax in which you "send a message to an object," the compiler does some undercover work for you. There's a secret first argument passed to the method **f( )**, and that argument is the reference to the object that's being manipulated. So the two method calls above become something like:

```
Banana.f(a,1);
Banana.f(b,2);
```

This is internal and you can't write these expressions and get the compiler to interchange them with **a.f( )**-style calls, but it gives you an idea of what's happening.

Suppose you're inside a method and you'd like to get the reference to the current object. Since that reference is passed *secretly* by the compiler, there's no identifier for it. However, for this purpose there's a keyword: **this**. The **this** keyword produces a reference to the object the method has been called for. You can treat this reference just like any other object reference. Keep in mind that if you're calling a method of your class from within another method of your class, you don't need to use **this**; you simply call the method. The current **this** reference is automatically used for the other method. Thus you can say:

```
class Apricot {
  int id;
  void pick() { /* ... */ }
  void pit() { pick(); id; /* ... */ }
}
```

Inside **pit( )**, you *could* say **this.pick( )** or **this.id** but there's no need to. The compiler does it for you automatically. The **this** keyword is used only for those special cases in which you need to explicitly use the reference to the current object (Visual Basic programmers may recognize the equivalent of the VB keyword **me)**. For example, it's often used in **return** statements when you want to return the reference to the current object:

```
//:c05:Leaf.cs
// Simple use of the "this" keyword.
using System;

public class Leaf {
  int i = 0;
  Leaf Increment() {
    i++;
    return this;
```

```
  }
  void Print() {
    Console.WriteLine("i = " + i);
  }
  public static void Main() {
    Leaf x = new Leaf();
    x.Increment().Increment().Increment().Print();
  }
} ///:~
```

Because **increment( )** returns the reference to the current object via the **this** keyword, multiple operations can easily be performed on the same object.

Another place where it's often used is to allow method parameters to have the same name as instance variables. Previously, we talked about the value of overloading methods so that the programmer only had to remember the one, most logical name. Similarly, the names of method parameters and the names of instance variables may also have a single logical name. C# allows you to use the **this** reference to disambiguate method variables (also called "stack variables") from instance variables. For clarity, you should use this capability only when the parameter is going to either be assigned to the instance variable (such as in a constructor) or when the parameter is to be compared against the instance variable. Method variables that have no correlation with same-named instance variables are a common source of lazy defects:

```
//:c05:Name.cs
using System;

class Name {
  string givenName;
  string surname;
  public Name(string givenName, string surname){
    this.givenName = givenName;
    this.surname = surname;
  }

  public bool perhapsRelated(string surname){
    return this.surname == surname;
  }

  public void printGivenName(){
    /* Legal, but unwise */
```

```
    string givenName = "method variable";
    Console.WriteLine("givenName is: " + givenName);
    Console.WriteLine(
      "this.givenName is: " + this.givenName);
  }

  public static void Main(){
    Name vanGogh = new Name("Vincent", "van Gogh");
    vanGogh.printGivenName();
    bool b = vanGogh.perhapsRelated("van Gogh");
    if (b) {
      Console.WriteLine("He's a van Gogh.");
    }
  }
}///:~
```

In the constructor, the parameters **givenName** and **surname** are assigned to the similarly-named instance variables and this is quite appropriate – calling the parameters **inGivenName** and **inSurname** (or worse, using parameter names such as **firstName** or **lastName** that do not correspond to the instance variables) would require explaining in the documentation. The **perhapsRelated( )** method shows the other appropriate use – the **surname** passed in is to be compared to the instance's **surname**. The **this.surname == surname** comparison in **perhapsRelated( )** might give you pause, because we've said that in general, the == operator compares addresses, not logical equivalence. However, the **string** class overloads the == operator so that it can be used for logically comparing values.

Unfortunately, the usage in **printGivenName( )** is also legal. Here, a variable called **givenName** is created on the stack; it has nothing to do with the instance variable also called **givenName**. It may be unlikely that someone would accidentally create a method variable called **givenName**, but you'd be amazed at how many **name**, **id**, and **flag**s one sees over the course of a career! It's another reason why meaningful variable names are important.

Sometimes you'll see code where half the variables begin with underscores and half the variables don't:

```
foo = _bar;
```

The intent is to use the prefix to distinguish between method variables that are created on the stack and go out of scope as soon as the method exits and variables that have longer lifespans. This is a bad idiom. For one thing, its origin had to do

with visibility, not storage, and C# has explicit and infinitely better visibility specifiers. For another, it's used inconsistently – almost as many people use the underscores for stack variables as use them for instance variables.

Sometimes you see code that prepends an 'm' to member variables names:

```
foo = mBar;
```

This isn't quite as bad as underscores. This type of naming convention is an offshoot of a C naming idiom called "Hungarian notation," that prefixes type information to a variable name (so strings would be **strFoo**). This is a great idea if you're programming in C and everyone who has programmed Windows has seen their share of variables starting with 'h', but the time for this naming convention has passed. One place where this convention continues is that interfaces (a type of object that has no implementation, discussed at length in Chapter 8) in the .NET Framework SDK are typically named with an initial "I" such as **IAccessible**.

If you want to distinguish between method and instance variables, use **this**:

```
foo = this.Bar;
```

It's object-oriented, descriptive, and explicit.

Calling constructors from constructors

When you write several constructors for a class, there are times when you'd like to call one constructor from another to avoid duplicating code. In C#, you can specify that another constructor execute before the current constructor. You do this using the ':' operator and the **this** keyword.

Normally, when you say **this**, it is in the sense of "this object" or "the current object," and by itself it produces the reference to the current object. In a constructor name, a colon followed by the **this** keyword takes on a different meaning: it makes an explicit call to the constructor that matches the specified argument list. Thus you have a straightforward way to call other constructors:

```
//:c05:Flower.cs
// Calling constructors with ": this."
using System;

public class Flower {
  int petalCount = 0;
  string s = "null";
  Flower(int petals) {
```

```
    petalCount = petals;
    Console.WriteLine(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(string ss) {
    Console.WriteLine(
      "Constructor w/ string arg only, s=" + ss);
    s = ss;
  }
  Flower(string s, int petals) : this(petals)
  //!, this(s) <- Can't call two base constructors!
  {
    this.s = s; // Another use of "this"
    Console.WriteLine("string & int args");
  }
  Flower() : this("hi", 47) {
    Console.WriteLine(
      "default constructor (no args)");
  }
  void Print() {
    Console.WriteLine(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void Main() {
    Flower x = new Flower();
    x.Print();
  }
}///:~
```

The constructor **Flower(String s, int petals)** shows that, while you can call one constructor using **this**, you cannot call two.

The meaning of static

With the **this** keyword in mind, you can more fully understand what it means to make a method **static**. It means that there is no **this** for that particular method. You cannot call non-**static** methods from inside **static** methods (although the reverse is possible), and you can call a **static** method for the class itself, without any object. In fact, that's primarily what a **static** method is for. It's as if you're creating the equivalent of a global function (from C). Except global functions are

not permitted in C#, and putting the **static** method inside a class allows it access to other **static** methods and **static** fields.

Some people argue that **static** methods are not object-oriented since they do have the semantics of a global function; with a **static** method you don't send a message to an object, since there's no **this**. This is probably a fair argument, and if you find yourself using a *lot* of static methods you should probably rethink your strategy. However, **static**s are pragmatic and there are times when you genuinely need them, so whether or not they are "proper OOP" should be left to the theoreticians. Indeed, even Smalltalk has the equivalent in its "class methods."

Cleanup: finalization and garbage collection

Programmers know about the importance of initialization, but often forget the importance of cleanup. After all, who needs to clean up an **int**? But with libraries, simply "letting go" of an object once you're done with it is not always safe. Of course, C# has the garbage collector to reclaim the memory of objects that are no longer used. Now consider a very unusual case. Suppose your object allocates "special" memory without using **new**. The garbage collector knows only how to release memory allocated *with* **new**, so it won't know how to release the object's "special" memory. To handle this case, C# provides a method called a *destructor* that you can define for your class. The destructor, like the constructor, shares the class name, but is prefaced with a tilde:

```
class MyClass{
   public MyClass(){ //Constructor }
   public ~MyClass(){ //Destructor }
}
```

C++ programmers will find this syntax familiar, but this is actually a dangerous mimic – the C# destructor has vastly different semantics, as you'll see. Here's how it's *supposed* to work. When the garbage collector is ready to release the storage used for your object, it will first call the object's destructor, and only on the next garbage-collection pass will it reclaim the object's memory. So if you choose to use the destructor, it gives you the ability to perform some important cleanup *at the time of garbage collection*.

This is a potential programming pitfall because some programmers, especially C++ programmers, because in C++ objects always get destroyed in a *deterministic* manner, whereas in C# the call to the destructor is *non-deterministic*. Since anything that needs special attention can't just be left around

to be cleaned up in a non-deterministic manner, the utility of C#'s destructor is severely limited. Or, put another way:

Clean up after yourself.

If you remember this, you will stay out of trouble. What it means is that if there is some activity that must be performed before you no longer need an object, you must perform that activity yourself. For example, suppose that you open a file and write stuff to it. If you don't explicitly close that file, it might not get properly flushed to the disk until the program ends.

You might find that the storage for an object never gets released because your program never nears the point of running out of storage. If your program completes and the garbage collector never gets around to releasing the storage for any of your objects, that storage will be returned to the operating system *en masse* as the program exits. This is a good thing, because garbage collection has some overhead, and if you never do it you never incur that expense.

What are destructors for?

A third point to remember is:

Garbage collection is only about memory.

That is, the sole reason for the existence of the garbage collector is to recover memory that your program is no longer using. So any activity that is associated with garbage collection, most notably your destructor method, must also be only about memory and its deallocation. Valuable resources, such as file handles, database connections, and sockets ought to be managed explicitly in your code, without relying on destructors.

Does this mean that if your object contains other objects, your destructor should explicitly release those objects? Well, no—the garbage collector takes care of the release of all object memory regardless of how the object is created. It turns out that the need for destructors is limited to special cases, in which your object can allocate some storage in some way other than creating an object. But, you might observe, everything in C# is an object so how can this be?

It would seem that C# has a destructor because of its support for unmanaged code, in which you can allocate memory in a C-like manner. Memory allocated in unmanaged code is not restored by the garbage collection mechanism. This is the one clear place where the C# destructor is necessary: when your class interacts with unmanaged code that allocates memory, place the code relating to cleaning up that memory in the destructor.

After reading this, you probably get the idea that you won't be writing destructors too often. Good. Destructors are called *non-deterministically* (that is, you cannot control when they are called), but valuable resources are too important to leave to happenstance.

The garbage collector *is* guaranteed to be called when your program ends, so you *may* include a "belts-and-suspender" last-chance check of any valuable resources that your object may wish to clean up. However, if the check ever finds the resource not cleaned up, don't pat yourself on the back – go in and fix your code so that the resource is cleaned up before the destructor is ever called!

Instead of a destructor, implement IDisposable.Dispose( )

The majority of objects don't use resources that need to be cleaned up. So most of the time, you don't worry about what happens when they "go away." But if you do use a resource, you should write a method called **Close( )** if the resource continues to exist after your use of it ends or **Dispose( )** otherwise. Most importantly, you should *explicitly* call the **Close( )** or **Dispose( )** method as soon as you no longer require the resource. This is just the principle of cleaning up after yourself.

If you rely on the garbage collector to manage resources, you can count on trouble:

```
//:c05:ValuableResource.cs
using System;
using System.Threading;

class ValuableResource {
  public static void Main(){
    useValuableResources();
    Console.WriteLine(
      "Valuable resources used and discarded");
    Thread.Sleep(10000);
    Console.WriteLine("10 seconds later...");
    //You would _think_ this would be fine
    ValuableResource vr = new ValuableResource();
  }

  static void useValuableResources(){
    for (int i = 0; i < MAX_RESOURCES; i++) {
```

```
      ValuableResource vr =
      new ValuableResource();
    }
  }

  static int idCounter;
  static int MAX_RESOURCES = 10;
  static int INVALID_ID = -1;
  int id;
  ValuableResource(){
    if (idCounter == MAX_RESOURCES) {
      Console.WriteLine(
        "No resources available");
      id = INVALID_ID;
    } else {
      id = idCounter++;
      Console.WriteLine(
        "Resource[{0}] Constructed", id);
    }
  }
  ~ValuableResource(){
    if (id == INVALID_ID) {
      Console.WriteLine("Things are awry!");
    } else {
      idCounter--;
      Console.WriteLine(
        "Resource[{0}] Destructed", id );
    }
  }
}///:~
```

In this example, the first thing that happens upon entering **Main( )** is the **useValuableResources( )** method is called. This is straightforward – the **MAX_RESOURCES** number of **ValuableResource** objects are created and then immediately allowed to "go away." In the **ValuableResource( )** constructor, the static **idCounter** variable is checked to see if it equals the **MAX_RESOURCES** value. If so, a "No resources available" message is written and the **id** of the **ValuableResource** is set to an invalid value (in this case, the **idCounter** is the source of the "scarce" resource which is "consumed" by the **id** variable). The **ValuableResource** destructor either outputs a warning message or decrements the **idCounter** (thus, making another "resource" available).

When **useValuableResources( )** returns, the system pauses for 10 seconds (we'll discuss **Thread.Sleep( )** in great detail in Chapter 16), and finally a new ValuableResource is created. It seems like that should be fine, since those created in **useValuableResources( )** are long gone. But the output tells a different story:

```
Resource[0] Constructed
Resource[1] Constructed
Resource[2] Constructed
Resource[3] Constructed
Resource[4] Constructed
Resource[5] Constructed
Resource[6] Constructed
Resource[7] Constructed
Resource[8] Constructed
Resource[9] Constructed
Valuable resources used and discarded
10 seconds later...
No resources available
Things are awry!
Resource[9] Destructed
Resource[8] Destructed
Resource[7] Destructed
Resource[6] Destructed
Resource[5] Destructed
Resource[4] Destructed
Resource[3] Destructed
Resource[2] Destructed
Resource[1] Destructed
Resource[0] Destructed
```

Even after ten seconds (an eternity in computing time), no id's are available and the final attempt to create a **ValuableResource** fails. The **Main( )** exits immediately after the "No resources available!" message is written. In this case, the CLR did a garbage collection as the program exited and the **~ValuableResource( )** destructors got called. In this case, they happen to be deleted in the reverse order of their creation, but the order of destruction of resources is yet another "absolutely not guaranteed" characteristic of garbage collection.

Worse, this is the output if one presses Ctl-C during the pause:

```
Resource[0] Constructed
```

```
Resource[1] Constructed
Resource[2] Constructed
Resource[3] Constructed
Resource[4] Constructed
Resource[5] Constructed
Resource[6] Constructed
Resource[7] Constructed
Resource[8] Constructed
Resource[9] Constructed
Valuable resources used and discarded
^C
D:\tic\chap4>
```

That's it. No cleanup. If the valuable resources were, say, network sockets or database connections or files or, well, anything that actually had any value, they'd be lost until you reboot (or some other process manages to restore their state by brute force, as can happen with files).

```
//:c05:ValuableResource2.cs
using System;
using System.Threading;

class ValuableResource {
  static int idCounter;
  static int MAX_RESOURCES = 10;
  static int INVALID_ID = -1;
  int id;

  ValuableResource(){
    if (idCounter == MAX_RESOURCES) {
      Console.WriteLine("No resources available");
      id = INVALID_ID;
    } else {
      id = idCounter++;
      Console.WriteLine(
        "Resource[{0}] Constructed", id);
    }
  }

  public void Dispose(){
    idCounter--;
    Console.WriteLine(
```

```
      "Resource[{0}] Destructed", id );
    if (id == INVALID_ID) {
      Console.WriteLine("Things are awry!");
    }
    GC.SuppressFinalize(this);
  }

  ~ValuableResource(){
    this.Dispose();
  }

  public static void Main(){
    UseValuableResources();
    Console.WriteLine(
      "Valuable resources used and discarded");
    Thread.Sleep(10000);
    Console.WriteLine("10 seconds later...");
    //This _is_ fine
    ValuableResource vr = new ValuableResource();
  }

  static void UseValuableResources(){
    for (int i = 0; i < MAX_RESOURCES; i++) {
      ValuableResource vr = new ValuableResource();
      vr.Dispose();
    }
  }
}///:~
```

We've moved the code that was previously in the destructor into a method called **Dispose( )**. Additionally, we've added the line:

```
GC.SuppressFinalize(this);
```

Which tells the Garbage Collector (the **GC** class object) not to call the destructor during garbage collection. We've kept the destructor, but it does nothing but call **Dispose( )**. In this case, the destructor is just a safety-net. It remains our responsibility to explicitly call **Dispose( )**, but if we don't and it so happens that the garbage collector gets first up, then our bacon is pulled out of the fire. Some argue this is worse than useless -- a method which isn't guaranteed to be called but which performs a critical function.

When ValuableResources2 is run, not only are there no problems with running out of resources, the **idCounter** never gets above zero!

The title of this section is: Destructors, IDisposable, and the using keywordInstead of a destructor, implement IDisposable.Dispose( ), but none of the examples actually implement this interface.

We've said that releasing valuable resources is the only task other than memory management that needs to happen during clean up. But we've also said that the call to the destructor is non-deterministic, meaning that the only guarantee about *when* it will be called is "before the application exits." So the main use of the destructor is as a last chance to call your **Dispose( )** method, which is where you should do the cleanup.

Why is **Dispose( )** the right method to use for special cleanup? Because the C# language has a way to guarantee that the **IDisposable.Dispose( )** method is called, even if something unusual happens. The technique uses object-oriented inheritance, which won't be discussed until Chapter 7. Further, to illustrate it, we need to throw an Exception, a technique which won't be discussed until Chapter 11! Rather than put off the discussion, though, it's important enough to present the technique here.

To ensure that a "cleanup method" is called as soon as possible:

1. Declare your class as implementing **IDisposable**
2. Implement **public void Dispose( )**
3. Place the vulnerable object inside a **using( )** block

The **Dispose( )** method will be called on exit from the **using** block. We're not going to go over this example in detail, since it uses so many as-yet-unexplored features, but the key is the block that follows the **using( )** declaration. When you run this code, you'll see that the **Dispose( )** method is called, then the code associated with the program leaving **Main( )**, and only then will the destructor be called!

```
//:c05:UsingCleanup.cs
using System;

class UsingCleanup : IDisposable {
```

```
  public static void Main(){
    try{
      UsingCleanup uc = new UsingCleanup();
      using(uc){
        throw new NotImplementedException();
      }
    }catch(NotImplementedException){
      Console.WriteLine("Exception ignored");
    }
    Console.WriteLine("Leaving Main( )");
  }

  UsingCleanup(){
    Console.WriteLine("Constructor called");
  }

  public void Dispose(){
    Console.WriteLine("Dispose called");
  }

  ~UsingCleanup(){
    Console.WriteLine("Destructor called");
  }
}///:~
```

How a garbage collector works

If you come from a programming language where allocating objects on the heap is expensive, you may naturally assume that C#'s scheme of allocating all reference types on the heap is expensive. However, it turns out that the garbage collector can have a significant impact on *increasing* the speed of object creation. This might sound a bit odd at first—that storage release affects storage allocation—but it means that allocating storage for heap objects in C# can be nearly as fast as creating storage on the stack in other languages.

For example, you can think of the C++ heap as a yard where each object stakes out its own piece of turf. This real estate can become abandoned sometime later and must be reused. In C#, the managed heap is quite different; it's more like a conveyor belt that moves forward every time you allocate a new object. This means that object storage allocation is remarkably rapid. The "heap pointer" is simply moved forward into virgin territory, so it's effectively the same as C++'s stack allocation. (Of course, there's a little extra overhead for bookkeeping but it's

nothing like searching for storage.) Yes, you heard right – allocation on the managed heap is *faster* than allocation within a C++-style unmanaged heap.

Now you might observe that the heap isn't in fact a conveyor belt, and if you treat it that way you'll eventually start paging memory a lot (which is a big performance hit) and later run out. The trick is that the garbage collector steps in and while it collects the garbage it compacts all the objects in the heap so that you've effectively moved the "heap pointer" closer to the beginning of the conveyor belt and further away from a page fault. The garbage collector rearranges things and makes it possible for the high-speed, infinite-free-heap model to be used while allocating storage.

To understand how this works, you need to get a little better idea of the way the Common Language Runtime garbage collector (GC) works. Garbage collection in the CLR (remember that memory management exists in the CLR "below" the level of the Common Type System, so this discussion equally applies to programs written in Visual Basic .NET, Eiffel .NET, and Python .NET as to C# programs) is based on the idea that any nondead object must ultimately be traceable back to a reference that lives either on the stack or in static storage. The chain might go through several layers of objects. Thus, if you start in the stack and the static storage area and walk through all the references you'll find all the live objects. For each reference that you find, you must trace into the object that it points to and then follow all the references in *that* object, tracing into the objects they point to, etc., until you've moved through the entire web that originated with the reference on the stack or in static storage. Each object that you move through must still be alive. Note that there is no problem with detached self-referential groups—these are simply not found, and are therefore automatically garbage. Also, if you trace to an object that has already been walked to, you do not have to re-trace it.

Having located all the "live" objects, the GC starts at the end of the managed heap and shifts the first live object in memory to be directly adjacent to the penultimate live object. This pair of live objects is then shifted to the next live object, the three are shifted en masse to the next, and so forth, until the heap is compacted.

Obviously, garbage collection is a lot of work, even on a modern, high-speed machine. In order to improve performance, the garbage collector refines the basic approach described here with *generations*.

The basic concept of generational garbage collection is that an object allocated recently is more likely to be garbage than an object which has already survived multiple passes of the garbage collector. So instead of walking the heap all the way from the stack or static storage, once the GC has run once, the collector may

assume that the previously compacted objects (the older generation) are all valid and only walk the most recently allocated part of the heap (the new generation).

Garbage collection is a favorite topic of researchers, and there will undoubtedly be innovations in GC that will eventually find their way into the field. However, garbage collection and computer power have already gotten to the stage where the most remarkable thing about GC is how transparent it is.

Member initialization

C# goes out of its way to guarantee that variables are properly initialized before they are used. In the case of variables that are defined locally to a method, this guarantee comes in the form of a compile-time error. So if you say:

```
void F() {
  int i;
  i++;
}
```

you'll get an error message that says that **i** is an unassigned local variable. Of course, the compiler could have given **i** a default value, but it's more likely that this is a programmer error and a default value would have covered that up. Forcing the programmer to provide an initialization value is more likely to catch a bug.

If a primitive is a data member of a class, however, things are a bit different. Since any method can initialize or use that data, it might not be practical to force the user to initialize it to its appropriate value before the data is used. However, it's unsafe to leave it with a garbage value, so each primitive data member of a class is guaranteed to get an initial value. Those values can be seen here:

```
//:c05:InitialValues.cs
// Shows default initial values.
using System;

class Measurement {
  bool t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
```

```
  double d;
  internal void Print() {
    Console.WriteLine(
      "Data type      Initial value\n" +
      "bool           " + t + "\n" +
      "char           [" + c + "] "+ (int)c +"\n"+
      "byte           " + b + "\n" +
      "short          " + s + "\n" +
      "int            " + i + "\n" +
      "long           " + l + "\n" +
      "float          " + f + "\n" +
      "double         " + d);
  }
}

public class InitialValues {
  public static void Main() {
    Measurement d = new Measurement();
    d.Print();
    /* In this case you could also say:
    new Measurement().print();
    */
  }
} ///:~
```

The output of this program is:

```
Data type      Initial value
boolean        Ffalse
char           [ ] 0
byte           0
short          0
int            0
long           0
float          0.0
double         0.0
```

The **char** value is a zero, which prints as a space.

You'll see later that when you define an object reference inside a class without initializing it to a new object, that reference is given a special value of **null** (which is a C# keyword).

You can see that even though the values are not specified, they automatically get initialized. So at least there's no threat of working with uninitialized variables.

Specifying initialization

What happens if you want to give a variable an initial value? One direct way to do this is simply to assign the value at the point you define the variable in the class. Here the field definitions in class **Measurement** are changed to provide initial values:

```
class Measurement {
  bool b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
  //…
```

You can also initialize nonprimitive objects in this same way. If **Depth** is a class, you can insert a variable and initialize it like so:

```
class Measurement {
  Depth o = new Depth();
  boolean b = true;
  // …
```

If you haven't given **o** an initial value and you try to use it anyway, you'll get a run-time error called an *exception* (covered in Chapter 11).

You can even call a static method to provide an initialization value:

```
class CInit {
  int i = InitI();
  //...
  static int InitI(){ //… }
}
```

This method can have arguments, but those arguments cannot be instance variables. Java programmers will note that this is more restrictive than Java's instance initialization, which can call non-static methods and use previously instantiated instance variables.

This approach to initialization is simple and straightforward. It has the limitation that *every* object of type **Measurement** will get these same initialization values. Sometimes this is exactly what you need, but at other times you need more flexibility.

Constructor initialization

The constructor can be used to perform initialization, and this gives you greater flexibility in your programming since you can call methods and perform actions at run-time to determine the initial values. There's one thing to keep in mind, however: you aren't precluding the automatic initialization, which happens before the constructor is entered. So, for example, if you say:

```
class Counter {
  int i;
  Counter() { i = 7; }
  // …
```

then **i** will first be initialized to 0, then to 7. This is true with all the primitive types and with object references, including those that are given explicit initialization at the point of definition. For this reason, the compiler doesn't try to force you to initialize elements in the constructor at any particular place, or before they are used—initialization is already guaranteed[1].

Order of initialization

Within a class, the order of initialization is determined by the order that the variables are defined within the class. The variable definitions may be scattered throughout and in between method definitions, but the variables are initialized before any methods can be called—even the constructor. For example:

```
//:c05:OrderOfInitialization.cs
// Demonstrates initialization order.
using System;

// When the constructor is called to create a
// Tag object, you'll see a message:
class Tag {
  internal Tag(int marker) {
```

[1] In contrast, C++ has the *constructor initializer list* that causes initialization to occur before entering the constructor body, and is enforced for objects. See *Thinking in C++, 2nd edition* (available at *www.BruceEckel.com*).

```
    Console.WriteLine("Tag(" + marker + ")");
  }
}

class Card {
  Tag t1 = new Tag(1); // Before constructor
  internal Card() {
    // Indicate we're in the constructor:
    Console.WriteLine("Card()");
    t3 = new Tag(33); // Reinitialize t3
  }
  Tag t2 = new Tag(2); // After constructor
  internal void F() {
    Console.WriteLine("F()");
  }
  Tag t3 = new Tag(3); // At end
}

public class OrderOfInitialization {
  public static void Main() {
    Card t = new Card();
    t.F(); // Shows that construction is done
  }
} ///:~
```

In **Card**, the definitions of the **Tag** objects are intentionally scattered about to prove that they'll all get initialized before the constructor is entered or anything else can happen. In addition, **t3** is reinitialized inside the constructor. The output is:

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

Thus, the **t3** reference gets initialized twice, once before and once during the constructor call. (The first object is dropped, so it can be garbage-collected later.) This might not seem efficient at first, but it guarantees proper initialization—what would happen if an overloaded constructor were defined that did *not* initialize **t3** and there wasn't a "default" initialization for **t3** in its definition?

Static data initialization

When the data is **static** the same thing happens; if it's a primitive and you don't initialize it, it gets the standard primitive initial values. If it's a reference to an object, it's **null** unless you create a new object and attach your reference to it.

If you want to place initialization at the point of definition, it looks the same as for non-**static**s. There's only a single piece of storage for a **static**, regardless of how many objects are created. But the question arises of when the **static** storage gets initialized. An example makes this question clear:

```
//:c05:StaticInitialization.cs
// Specifying initial values in a
// class definition.
using System;

class Bowl {
  internal Bowl(int marker) {
    Console.WriteLine("Bowl(" + marker + ")");
  }
  internal void F(int marker) {
    Console.WriteLine("F(" + marker + ")");
  }
}

class Table {
  static Bowl b1 = new Bowl(1);
  internal Table() {
    Console.WriteLine("Table()");
    b2.F(1);
  }
  internal void F2(int marker) {
    Console.WriteLine("F2(" + marker + ")");
  }
  static Bowl b2 = new Bowl(2);
}

class Cupboard {
  Bowl b3 = new Bowl(3);
  static Bowl b4 = new Bowl(4);
  internal Cupboard() {
    Console.WriteLine("Cupboard()");
```

```
    b4.F(2);
  }
  internal void F3(int marker) {
    Console.WriteLine("F3(" + marker + ")");
  }
  static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
  public static void Main() {
    Console.WriteLine(
      "Creating new Cupboard() in main");
    new Cupboard();
    Console.WriteLine(
      "Creating new Cupboard() in main");
    new Cupboard();
    t2.F2(1);
    t3.F3(1);
  }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl allows you to view the creation of a class, and **Table** and **Cupboard** create **static** members of **Bowl** scattered through their class definitions. Note that **Cupboard** creates a non-**static Bowl b3** prior to the **static** definitions. The output shows what happens:

```
Bowl(1)
Bowl(2)
Table()
fF(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
fF(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
fF(2)
Creating new Cupboard() in main
```

```
Bowl(3)
Cupboard()
fF(2)
f2F2(1)
f3F3(1)
```

The **static** initialization occurs only if it's necessary. If you don't create a **Table** object and you never refer to **Table.b1** or **Table.b2**, the **static Bowl b1** and **b2** will never be created. However, they are initialized only when the *first* **Table** object is created (or the first **static** access occurs). After that, the **static** objects are not reinitialized.

The order of initialization is **static**s first, if they haven't already been initialized by a previous object creation, and then the non-**static** objects. You can see the evidence of this in the output.

It's helpful to summarize the process of creating an object. Consider a class called **Dog**:

1. The first time an object of type **Dog** is created, *or* the first time a **static** method or **static** field of class **Dog** is accessed, the C# runtime must locate the assembly in which **Dog**'s class definition is stored.
2. As the **Dog** class is loaded (creating a **Type** object, which you'll learn about later), all of its **static** initializers are run. Thus, **static** initialization takes place only once, as the **Type** object is loaded for the first time.
3. When you create a **new Dog( )**, the construction process for a **Dog** object first allocates enough storage for a **Dog** object on the heap.
4. This storage is wiped to zero, automatically setting all the primitives in that **Dog** object to their default values (zero for numbers and the equivalent for **bool** and **char**) and the references to **null**.
5. Any initializations that occur at the point of field definition are executed.
6. Constructors are executed. As you shall see in Chapter 7, this might actually involve a fair amount of activity, especially when inheritance is involved.

Static constructors

C# allows you to group other **static** initializations inside a special "**static** constructor." It looks like this:

```
class Spoon {
  static int i;
  static Spoon(){
    i = 47;
  }
  // …
}
```

This code, like other **static** initializations, is executed only once, the first time you make an object of that class *or* the first time you access a **static** member of that class (even if you never make an object of that class). For example:

```
//:c05:StaticConstructor.cs
// Explicit static initialization
// with static constructor
using System;

class Cup {
  internal Cup(int marker) {
    Console.WriteLine("Cup(" + marker + ")");
  }
  internal void F(int marker) {
    Console.WriteLine("f(" + marker + ")");
  }
}

class Cups {
  internal static Cup c1;
  static Cup c2;
  static Cups(){
    Console.WriteLine(
      "Inside static Cups() constructor");
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  Cups() {
    Console.WriteLine("Cups()");
  }
}

public class ExplicitStatic {
  public static void Main() {
```

```
    Console.WriteLine("Inside Main()");
    Cups.c1.F(99);  // (1)
  }
  // static Cups x = new Cups();  // (2)
  // static Cups y = new Cups();  // (2)
} ///:~
```

The **static** constructor for **Cups** run when either the access of the **static** object **c1** occurs on the line marked (1), or if line (1) is commented out and the lines marked (2) are uncommented. If both (1) and (2) are commented out, the **static** constructor for **Cups** never occurs. Also, it doesn't matter if one or both of the lines marked (2) are uncommented; the static initialization only occurs once.

Array initialization

Initializing arrays in C is error-prone and tedious. C++ uses *aggregate initialization* to make it much safer[2]. C# has no "aggregates" like C++, since everything is an object in C#. It does have arrays, and these are supported with array initialization.

An array is simply a sequence of either objects or primitives, all the same type and packaged together under one identifier name. Arrays are defined and used with the square-brackets *indexing operator* **[ ]**. To define an array you simply follow your type name with empty square brackets:

```
int[] a1;
```

This is a little different from C and C++, but is a sensible improvement, since it says that the type is "an **int** array."

The compiler doesn't allow you to tell it how big the array is. This brings us back to that issue of "references." All that you have at this point is a reference to an array, and there's been no space allocated for the array. To create storage for the array you must write an initialization expression. For arrays, initialization can appear anywhere in your code, but you can also use a special kind of initialization expression that must occur at the point where the array is created. This special initialization is a set of values surrounded by curly braces. The storage allocation (the equivalent of using **new**) is taken care of by the compiler in this case. For example:

---

[2] See *Thinking in C++, 2nd edition* for a complete description of C++ aggregate initialization.

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

So why would you ever define an array reference without an array?

```
int[] a2;
```

Well, it's possible to assign one array to another in C#, so you can say:

```
a2 = a1;
```

What you're really doing is copying a reference, as demonstrated here:

```
//:c05:Arrays.cs
// Arrays of primitives.
using System;

public class Arrays {
  public static void Main() {
    int[] a1 = { 1, 2, 3, 4, 5};
    int[] a2;
    a2 = a1;
    for (int i = 0; i < a2.Length; i++)
      a2[i]++;
    for (int i = 0; i < a1.Length; i++)
      Console.WriteLine("a1[" + i + "] = " + a1[i]);
  }
} ///:~
```

You can see that **a1** is given an initialization value while **a2** is not; **a2** is assigned later—in this case, to another array.

There's something new here: all arrays have a property (whether they're arrays of objects or arrays of primitives) that you can query—but not change—to tell you how many elements there are in the array. This member is **Length**. Since arrays in C#, as in Java and C, start counting from element zero, the largest element you can index is **Length - 1**. If you go out of bounds, C and C++ quietly accept this and allow you to stomp all over your memory, which is the source of many infamous bugs. However, C# protects you against such problems by causing a run-time error (an *exception*, the subject of Chapter 11) if you step out of bounds. Of course, checking every array access costs time and code, which means that array accesses might be a source of inefficiency in your program if they occur at a critical juncture. Sometimes the JIT can "precheck" to ensure that all index values in a loop will never exceed the array bounds, but in general, array access pays a small performance price. By explicitly moving to "unsafe" code (discussed in Chapter 10), bounds checking can be turned off.

What if you don't know how many elements you're going to need in your array while you're writing the program? You simply use **new** to create the elements in the array. Here, **new** works even though it's creating an array of primitives (**new** won't create a nonarray primitive):

```
//:c05:ArrayNew.cs
// Creating arrays with new.
using System;

public class ArrayNew {
  static Random rand = new Random();

  public static void Main() {
    int[] a;
    a = new int[rand.Next(20) + 1];
    Console.WriteLine("length of a = " + a.Length);
    for (int i = 0; i < a.Length; i++)
      Console.WriteLine("a[" + i + "] = " + a[i]);
  }
}
///:~
```

Since the size of the array is chosen at random, it's clear that array creation is actually happening at run-time. In addition, you'll see from the output of this program that array elements of primitive types are automatically initialized to "empty" values. (For numerics and **char**, this is zero, and for **bool**, it's **false**.)

If you're dealing with an array of nonprimitive objects, you must always use **new**. Here, the reference issue comes up again because what you create is an array of references. Consider the wrapper type **IntHolder,** which is a class and not a primitive:

```
//:c05:ArrayClassObj.cs
// Creating an array of nonprimitive objects.
using System;

class IntHolder {
  int i;
  internal IntHolder(int i){
    this.i = i;
  }

  public override string ToString(){
```

```
    return i.ToString();
  }
}

public class ArrayClassObj {
  static Random rand = new Random();

  public static void Main() {
    IntHolder[] a = new IntHolder[rand.Next(20) + 1];
    Console.WriteLine("length of a = " + a.Length);
    for (int i = 0; i < a.Length; i++) {
      a[i] = new IntHolder(rand.Next(500));
      Console.WriteLine("a[" + i + "] = " + a[i]);
    }
  }
} ///:~
```

Here, even after **new** is called to create the array:

```
IntHolder[] a = new IntHolder[rand.Next(20) + 1];
```

it's only an array of references, and not until the reference itself is initialized by creating a new **IntHolder** object is the initialization complete:

```
a[i] = new IntHolder(rand.Next(500));
```

If you forget to create the object, however, you'll get an exception at run-time when you try to read the empty array location.

The **IntHolder** method **ToString( )** is marked with the **override** keyword. This will be discussed in more detail later, but the short explanation is that this is an object-oriented refinement of a **ToString( )** method defined in some class that is an "ancestor" to **IntHolder** (in fact, the **ToString( )** method is defined in the class **Object**, which is the ancestor to *all* classes).

It's also possible to initialize arrays of objects using the curly-brace-enclosed list. There are two forms:

```
//:c05:ArrayInit.cs
// Array initialization.

class IntHolder {
  int i;
  internal IntHolder(int i){
    this.i = i;
```

```
  }
}

public class ArrayInit {
  public static void Main() {
    IntHolder[] a = {
      new IntHolder(1),
      new IntHolder(2),
      new IntHolder(3),
    };

    IntHolder[] b = new IntHolder[] {
      new IntHolder(1),
      new IntHolder(2),
      new IntHolder(3),
    };
  }
} ///:~
```

This is useful at times, but it's more limited since the size of the array is determined at compile-time. The final comma in the list of initializers is optional. (This feature makes for easier maintenance of long lists.)

The params method modifier

An unusual use of arrays is C#'s **params** method argument modifier. This modifier, when applied to the last parameter of a method, specifies that the method can be called with any number of arguments of the specified type. In this case, a **Burger** can be created with any number of beef patties:

```
//:c05:Burger.cs
using System;

class Patty {
}
class Burger {
  Burger(bool cheese, params Patty[] patties){
    foreach(Patty p in patties){
      if (cheese) {
        Console.WriteLine("Cheeseburger!");
      } else {
        Console.WriteLine("Hamburger!");
```

```
      }
    }
    Console.WriteLine("You want fries with that?");
  }

  public static void Main(){
    Burger noMeat = new Burger(false);
    Burger petite = new Burger(false, new Patty());
    //Double cheeseburger
    Burger doubleDouble =
      new Burger(true, new Patty(), new Patty());
    //Heart attack
    Burger fourByFour =
      new Burger(true, new Patty(), new Patty(),
        new Patty(), new Patty());
  }
}///:~
```

The interesting part is in **Burger.Main( )**, which shows the **Burger** constructor being called with various amounts of **Patty**s (even no patties).

The **params** modifier is how the **String.Format( )** method and **Console.WriteLine( )** allow us to write lines such as:

```
   String.Format("f{0}c{1}t{2}{3}{4}sl{5}"
                 ,'a','e','i','o','u','y');
```

Multidimensional arrays

C# allows you to easily create multidimensional arrays:

```
//:c05:MultiDimArray.cs
// Creating multidimensional arrays.
using System;

class IntHolder {
  int i;
  internal IntHolder(int i) {
    this.i = i;
  }

  public override string ToString() {
    return i.ToString();
```

```
  }
}

public class MultiDimArray {
  static Random rand = new Random();

  static void Prt(string s) {
    Console.WriteLine(s);
  }

  public static void Main() {
    int[,] a1 = {
      { 1, 2, 3,},
      { 4, 5, 6,},
    };
    Prt("a1.Length = " + a1.Length);
    Prt(" == " + a1.GetLength(0)
      + " * " + a1.GetLength(1));
    for (int i = 0; i < a1.GetLength(0); i++)
      for (int j = 0; j < a1.GetLength(1); j++)
        Prt("a1[" + i + "," + j
          + "] = " + a1[i, j]);
    // 3-D rectangular array:
    int[,,] a2 = new int[2, 2, 4];
    for (int i = 0; i < a2.GetLength(0); i++)
      for (int j = 0; j < a2.GetLength(1); j++)
        for (int k = 0; k < a2.GetLength(2);
            k++)
          Prt("a2[" + i + ","
            + j + "," + k
            + "] = " + a2[i,j,k]);
    // Jagged array with varied-Length vectors:
    int[][][] a3 = new int[rand.Next(7) + 1][][];
    for (int i = 0; i < a3.Length; i++) {
      a3[i] = new int[rand.Next(5) + 1][];
      for (int j = 0; j < a3[i].Length; j++)
        a3[i][j] = new int[rand.Next(5) + 1];
    }
    for (int i = 0; i < a3.Length; i++)
      for (int j = 0; j < a3[i].Length; j++)
        for (int k = 0; k < a3[i][j].Length;
```

```
            k++)
          Prt("a3[" + i + "]["
            + j + "][" + k
            + "] = " + a3[i][j][k]);
    // Array of nonprimitive objects:
    IntHolder[,] a4 = {
      { new IntHolder(1), new IntHolder(2)},
      { new IntHolder(3), new IntHolder(4)},
      { new IntHolder(5), new IntHolder(6)},
    };
    for (int i = 0; i < a4.GetLength(0); i++)
      for (int j = 0; j < a4.GetLength(1); j++)
        Prt("a4[" + i + "," + j
          + "] = " + a4[i,j]);
    IntHolder[][] a5;
    a5 = new IntHolder[3][];
    for (int i = 0; i < a5.Length; i++) {
      a5[i] = new IntHolder[3];
      for (int j = 0; j < a5[i].Length; j++) {
        a5[i][j] = new IntHolder(i*j);
      }
    }
    for (int i = 0; i < a5.GetLength(0); i++) {
      for (int j = 0; j < a5[i].Length; j++) {
        Prt("a5[" + i + "][" + j
          + "] = " + a5[i][j]);
      }
    }
  }
} ///:~
```

The code used for printing uses **Length** so that it doesn't depend on fixed array sizes.

The first example shows a multidimensional *rectangular array* of primitives. You delimit each vector in the array with curly braces:

```
    int[,] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
```

Each comma in the index moves you into the next level of the array.

The second example shows a three-dimensional rectangular array allocated with **new**. Here, the whole array is allocated at once:

```
int[,,] a2 = new int[2, 2, 4];
```

In a rectangular array, each vector that makes up the array is a fixed size, and therefore the array is itself a fixed size, in this case, an array of precisely the size needed to hold 16 (2 * 2 * 4) integers.

The third example shows a different type of array, a *jagged array* in which each vector in the arrays that make up the matrix can be of any length:

```
int[][][] a3 = new int[rand.Next(7) + 1][][];
for(int i = 0; i < a3.Length; i++) {
  a3[i] = new int[rand.Next(5)][];
  for(int j = 0; j < a3[i].Length; j++)
    a3[i][j] = new int[rand.Next(5)];
}
```

The first **new** creates an array with a random-length first element and the rest undetermined. The second **new** inside the **for** loop fills out the elements but leaves the third index undetermined until you hit the third **new**.

You will see from the output that array values are automatically initialized to zero if you don't give them an explicit initialization value.

You can deal with arrays of nonprimitive objects in a similar fashion, which is shown in the fourth example, demonstrating the ability to collect many **new** expressions with curly braces:

```
IntHolder[,] a4 = {
  { new IntHolder(1), new IntHolder(2)},
  { new IntHolder(3), new IntHolder(4)},
  { new IntHolder(5), new IntHolder(6)},
};
```

The fifth example shows how an array of nonprimitive objects can be built up piece by piece:

```
IntHolder[][] a5;
a5 = new IntHolder[3][];
for(int i = 0; i < a5.length; i++) {
  a5[i] = new IntHolder[3];
  for(int j = 0; j < a5[i].length; j++)
    a5[i][j] = new IntHolder(i*j);
```

```
    }
```

The **i*j** is just to put an interesting value into the **IntHolder**.

What a difference a rectangle makes

The addition of rectangular arrays to C# is one of a few different language features that have the potential to make C# a great language for numerically intensive computing. With jagged arrays (arrays of the form **Object[ ][ ]**), it's impossible for an optimizer to make assumptions about memory allocation. A jagged array may have multiple rows pointing to the same base array, unallocated rows, and cross-references.

```
double[5][] myArray = new double[5][];
myArray[0] = new double[2];
myArray[1] = myArray[0];
myArray[2] = new double[1];
myArray[4] = new double[4];
```

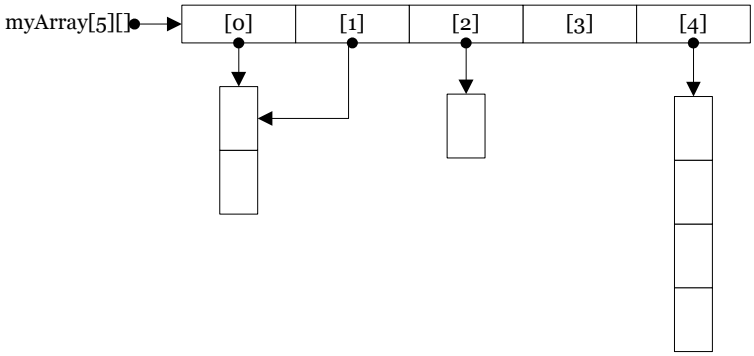


Figure 5-1: Jagged arrays can have complex relationships to physical memory

A rectangular array, on the other hand, is a contiguous block:

```
double myArray[5,4] = new double[5,4];
```

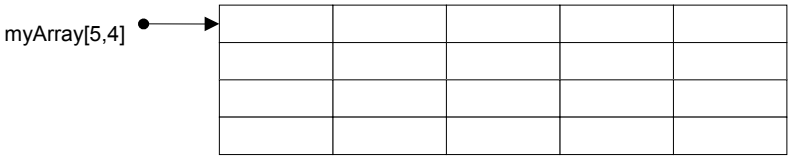


Figure 5-2: Rectangular arrays are contiguous blocks of memory

As you can see from Figures 5-1 and 5-2, jagged arrays are best thought of as "references to references" while a rectangular array can be safely thought of as a

"grid of references." Since physical RAM is not a grid at all, but continuous, a rectangular array is *really* a single contiguous chunk of memory. Jagged arrays are more flexible in terms of efficiently storing references without having to copy them back and forth, but rectangular arrays are perhaps a tiny bit easier to initialize and use. In addition, several optimizing techniques are harder to do with jagged arrays than with rectangular. When researchers at IBM added rectangular arrays to Java, they speeded up some numerical benchmarks by factors close to 50! So far, the C# optimizer doesn't take advantage of such possibilities, although it does run somewhat faster than Java on Cholesky multiplication[3].

Summary

This seemingly elaborate mechanism for initialization, the constructor, should give you a strong hint about the critical importance placed on initialization in the language. As Stroustrup was designing C++, one of the first observations he made about productivity in C was that improper initialization of variables causes a significant portion of programming problems. These kinds of bugs are hard to find, and similar issues apply to improper cleanup. Because constructors allow you to *guarantee* proper initialization and cleanup (the compiler will not allow an object to be created without the proper constructor calls), you get complete control and safety.

In C++, destruction is quite important because objects created with **new** must be explicitly destroyed. In C#, the garbage collector automatically releases the memory for all objects, so the equivalent cleanup method in C# isn't necessary much of the time. In cases where you don't need destructor-like behavior, C#'s garbage collector greatly simplifies programming, and adds much-needed safety in managing memory. However, the garbage collector does add a run-time cost, the expense of which is difficult to put into perspective because of the other performance ramifications of the IL and CLR approach to binary files.

Because of the guarantee that all objects will be constructed, there's actually more to the constructor than what is shown here. In particular, when you create new classes using either *composition* or *inheritance* the guarantee of construction also

---

[3] The original article is *The Ninja Project*, Moreira et al., Communications of the ACM 44(10), Oct 2001. For more on C# performance, including a port of some of the benchmarks used by Moreira from Java to C#, see *http://www.ThinkingIn.Net /performance.html*

holds, and some additional syntax is necessary to support this. You'll learn about composition, inheritance, and how they affect constructors in future chapters.

Exercises

1. Create a class with a default constructor (one that takes no arguments) that prints a message. Create an object of this class.
2. Add an overloaded constructor to Exercise 1 that takes a string argument and prints it along with your message.
3. Create an array of object references of the class you created in Exercise 2, but don't actually create objects to assign into the array. When you run the program, notice whether the initialization messages from the constructor calls are printed.
4. Complete Exercise 3 by creating objects to attach to the array of references.
5. Create an array of string objects and assign a string to each element. Print the array using a **foreach** loop.
6. Create a class called **Dog** with an overloaded **Bark( )** method. This method should be overloaded based on various primitive data types, and print different types of barking, howling, etc., depending on which overloaded version is called. Write a **Main( )** that calls all the different versions.
7. Modify Exercise 6 so that two of the overloaded methods have two arguments (of two different types), but in reversed order relative to each other. Verify that this works.
8. Create a class without a constructor, and then create an object of that class in **Main( )** to verify that the default constructor is automatically synthesized.
9. Create a class with two methods. Within the first method, call the second method twice: the first time without using this, and the second time using this.
10. Create a class with a destructor that prints a message. In **Main( )**, create an object of your class. Explain the behavior of your program.

11. Modify Exercise 11 so that the object is created within a method other than **Main( )**. Modify your class so that it implements **IDisposable** and the **Dispose( )** method is called before **Main( )** exits.

12. Create a class called **Tank** that can be filled and emptied, and has a death condition that it must be empty when the object is cleaned up. Write a **Dispose( )** that verifies this death condition. In **Main( )**, test the possible scenarios that can occur when your **Tank** is used.

13. Create a class containing an **int** and a **char** that are not initialized, and print their values to verify that C# performs default initialization.

14. Create a class containing a declared but uninitialized **String** reference. Demonstrate that this reference is initialized by C# to **null**.

15. Create a class with a string field that is initialized at the point of definition, and another one that is initialized by the constructor. What is the difference between the two approaches?

16. Create a class with a static string field that is initialized at the point of definition, and another one that is initialized by a static constructor. Add a static method that prints both fields and demonstrates that they are both initialized before they are used.

17. Write a method that creates and initializes a two-dimensional array of double. The size of the array is determined by the arguments of the method, and the initialization values are a range determined by beginning and ending values that are also arguments of the method. Create a second method that will print the array generated by the first method. In **Main( )** test the methods by creating and printing several different sizes of arrays.

18. Repeat Exercise 17 for a three-dimensional array.

19. Comment the line marked (1) in **StaticConstructor.cs** and verify that the static constructor is not called. Now uncomment one of the lines marked (2) and verify that the static constructor is called. Now uncomment the other line marked (2) and verify that static construction only occurs once.

20. Referring back to the robotic party servant exercises from previous chapters, describe the initialization and cleanup required for each of your classes. Find at least one class that represents a “valuable resource” for the party that is not in infinite supply.

21. For the "valuable resource" discovered in exercise 20, implement the **Dispose( )** method of the class. Write a program that creates and consumes this resource and demonstates proper cleanup.

22. Try to find and implement a class in the party domain that requires static initialization.

23. Try to find and implement a class in the party domain that uses an array initialization

24. You should now have at least 5 programs in the party domain. Eliminate common code by refactoring them into utility functions. Confirm that *all* programs in the party domain continue to function!

6: Hiding the Implementation

A primary consideration in object-oriented design is "separating the things that change from the things that stay the same."

This is particularly important for libraries. The user (*client programmer*) of that library must be able to rely on the part they use, and know that they won't need to rewrite code if a new version of the library comes out. On the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client programmer's code won't be affected by those changes.

This can be achieved through convention. For example, the library programmer must agree to not remove existing methods when modifying a class in the library, since that would break the client programmer's code. The reverse situation is thornier, however. In the case of a data member, how can the library creator know which data members have been accessed by client programmers? This is also true with methods that are only part of the implementation of a class, and not meant to be used directly by the client programmer. But what if the library creator wants to rip out an old implementation and put in a new one? Changing any of those members might break a client programmer's code. Thus the library creator is in a strait jacket and can't change anything.

To solve this problem, C# provides *access specifiers* to allow the library creator to say what is available to the client programmer and what is not. The levels of access control from "most access" to "least access" are **public, protected internal**, **protected**, **internal**, and **private**. From the previous paragraph you might think that, as a library designer, you'll want to keep everything as "private" as possible, and expose only the methods that you want the client programmer to use. This is exactly right, even though it's often counterintuitive for people who program in other languages (especially C) and are used to accessing everything without restriction. By the end of this chapter you should be convinced of the value of access control in C#.

The concept of a library of components and the control over who can access the components of that library is not complete, however. There's still the question of how the components are bundled together into a cohesive library unit. This is controlled by the **namespace** keyword for creating related names and by bundling related classes into assemblies. The access specifiers are affected by whether a class is in the same assembly or in a separate assembly. Before we discuss assemblies though, we need to learn about namespaces. Then you'll be able to understand the complete meaning of the access specifiers and move onto multiple assemblies.

Organizing with namespaces

A namespace is what you get when you use the **using** keyword to bring in the classes named in n entire library, such as

```
using System.Collections;
```

This makes visible to your code the entire System.Collections library that's part of the standard .NET Framework SDK distribution. Since, for example, the class **ArrayList** is in **System.Collections**, you can now either specify the full name **System.Collections.ArrayList** (which you can do without the **using** statement), or you can simply say **ArrayList** (because of the **using**).

Namespaces exist to insulate related classes so that they can see each other, but non-related classes cannot see all of the classes. A method **F( )** inside a class **A** will not clash with an **F( )** that has the same signature (argument list) in class **B**. But what about the class names? Suppose you create a **Stack** class that is installed on a machine that already has a **Stack** class that's written by someone else? With C# and the Internet, this can happen without the user knowing it, since class assemblies can be downloaded automatically in the process of running a .NET program.

This potential clashing of names is why it's important to have complete control over the name spaces in C#, and to be able to create a completely unique name regardless of the constraints of the Internet.

So far, most of the examples in this book have existed in a single file and have been designed for local use, and haven't bothered with namespaces. (In this case the class name is placed in the "default namespace.") This is certainly an option, and for simplicity's sake this approach will be used whenever possible throughout the rest of this book. However, if you're planning to create libraries or programs that are friendly to other C# programs on the same machine, you must think about preventing class name clashes.

When you create a source-code file for C#, it's commonly called a *compilation unit* (sometimes a *translation unit*). By convention, each compilation unit has a name ending in **.cs**. A compilation unit in C# may contain as many types as desired. The compiler (**csc.exe**) translates one or more compilation unit into an *assembly*. An assembly consists of some number of **public** classes that are available for use by other assemblies and some amount of non-visible support classes.

In contrast with Java, C# does not enforce a strict correspondence between compilation units, public classes, namespaces, and assemblies. However, it is generally a good idea to:

- Restrict a compilation unit to a single namespace
- Put only one **public** class in a single compilation unit
- Put the compilation units for a single namespace in a single source directory

These are good ideas for a couple of different reasons. For one thing, once a program gets to a medium size and has, say, a few dozen thousand lines of code, even the very fast C# compiler takes a significant amount of time to recompile *everything*. Second, the compilation unit is the natural unit of source code control and testing. While it's possible to create systems that are intelligent about merging disparate changes and only testing changed classes, these are by far more complex and error-prone than systems that are based on simply comparing the timestamps on the compilation unit files!

Just because you've created a class in a namespace and compiled it into an assembly does not automatically make it available to other software. For instance, if you write:

```
namespace MyNamespace{
   public class MyClass{
      //…etc…
   }
}
```

and save it to a file **MyClass.cs**, you would compile with:

```
csc /target:library MyClass.cs
```

which would create an assembly called **MyClass.DLL**. To use this assembly from another compilation unit, you would write something of this form:

```
using MyNamespace;
```

```
namespace MyNewNamespace{
   class MyNewClass{
     //Class referenced in "using"
     MyClass c = new MyClass();
     //… etc …
   }
}
```

but you would have to compile **MyNewClass.cs** with an explicit reference to the assembly in which **MyClass** is stored:

```
csc /reference:MyClass.dll MyNewClass.cs
```

Otherwise you would receive a compilation error that "The type or namespace 'MyNamespace' could not be found."

In addition, when the first object of type **MyNewClass** is created *at runtime*, it must be able to find the **MyClass.dll** assembly. The easiest way to do this is to place a copy of **MyClass.dll** in the same directory as the assembly that contains **MyNewClass**. One of the greatest differences between .NET and previous versions of Windows is support for "XCOPY deployment." This means that, except for niceties like creating shortcuts and items on the **Start** menu, you can deploy a .NET application simply by copying files.

Creating unique names

You might observe that if you wanted your library assembly **MyClass.dll** to be used by two or more applications, XCOPY deployment might not be the best solution. It would be better if you installed your library into some location that both applications (heck, *all* applications) automatically checked. As a Windows user, this would naturally make you think of the Registry and subdirectories of the Windows directory.

If you are a Windows programmer, you will also be aware of the problems this can raise. What if another company ships an assembly named **MyClass.dll** or someone accidentally copies an old version on top of your most recent one? These are the causes of "DLL Hell," that Windows condition in which installing a new application causes problems in others.

To avoid DLL Hell, .NET provides an entirely new system of sharing assemblies. This system has two components: qunique names based on public-key cryptography and a set of sub-directories known as the *Global Assembly Cache* (GAC). Each deployed version of the assembly gets its own unique identity; the

system-level tools that manipulate the GAC assure that uniquely identified assemblies do not overwrite one another. In addition, the choice to use cryptography to create the unique names means that, in addition, the tools can guarantee the integrity of the bits in the assembly. The process by which assemblies are signed and installed into the GAC is covered in Chapter 13.

While cryptographically verified *strong names* assure binary stability, you should still strive to create unique namespaces to aid in the organization of your work. The naming convention for namespaces is ***CompanyName.ProjectName.SubSystemName***. For instance, you might create your own **ArrayList** in a **Collections** subsystem:

```
//compiled with csc /target:library
using System;
namespace Thinkingin.CSharp.Collections{
  public class ArrayList{
    public ArrayList(){
       Console.WriteLine
             ("ThinkingIn.CSharp.Collections.ArrayList");
    }
  }
}
```

The second file must reference this namespace before use:

```
using Thinkingin.CSharp.Collections;

namespace usesanother{
  class UsesSpecialized{
    public static void Main(){
      ArrayList al = new ArrayList();
      //Can still explicitly reference other
      System.Collections.ArrayList realList =
        new System.Collections.ArrayList();
      realList.Add(
        "Oh! It's a real collection class!");
    }
  }
}
```

The compiler requires both a **using** statement in the source code and a **/reference** switch on the command-line to bring in libraries that are not in **mscorlib.dll** ("Microsoft .NET Core Libraries"). The referenced assembly must

be in the path *both* at compile time and at runtime (unless it is loaded from the GAC, as described in Chapter 13).

Collisions

What happens if two libraries are imported via **using** and they include the same names? For example, suppose a program does this:

```
using ThinkingIn.CSharp.Collections;
using System.Collections;
```

Since **System.Collections** also contains an **ArrayList** class, this causes a potential collision. However, as long as the collision does not actually occur, everything is OK – this is good because otherwise you might end up doing a lot of typing to prevent collisions that would never happen.

The collision *does* occur if you now try to make an ArrayList:

```
ArrayList al = new ArrayList();
```

Which **ArrayList** class does this refer to? The compiler can't know, and the reader can't know either. So the compiler complains and forces you to *disambiguate* the reference. If you want a standard .NET **ArrayList**, for example, you must say:

```
System.Collections.ArrayList al =
   new System.Collections.ArrayList();
```

This (along with the assembly references specified at the command-line) completely specifies which class you mean; the compiler can allow both **using** statements to coexist.

Using #define to change behavior

A compilation feature that C# shares with C is the ability to change the behavior of the compiler based on *meta-commands* embedded in the code or specified as part of the compiler's command-line. A common use for this feature is to enable or disable debugging code. During development, debugging code that performs costly verification or informational output is enabled; when release nears, it is disabled. Here's an example:

```
//:c06:TestDebug.cs
// Demonstrating conditional compilation.
// Comment or uncomment the following to change behavior:
#define DEBUG
using System;
```

```
using System.Diagnostics;

public class Assert {
  private static void PErr(string s){
    Console.WriteLine(s);
  }
  [Conditional("DEBUG")]
  public static void True(bool exp){
    if (!exp) PErr("Assertion failed");
  }
  [Conditional("DEBUG")]
  public static void False(bool exp){
    if (exp) PErr("Assertion failed");
  }
  [Conditional("DEBUG")]
  public static void True(bool exp, string msg){
    if (!exp) PErr(msg);
  }
  [Conditional("DEBUG")]
  public static void False(bool exp, string msg){
    if (exp) PErr(msg);
  }
}

public class TestDebug {
  public static void Main() {
    Assert.True((2 + 2) == 5);
    Assert.False((1 + 1) == 2);
    Assert.True((2 + 2) == 5, "2 + 2 == 5");
    Assert.False((1 + 1) == 2, "1 +1 != 2");
  }
} //:~
```

By commenting or uncommenting **#define DEBUG**, you change your code from the debug version to the production version. This technique can be used for any kind of conditional code.

The use of some kind of **#define DEBUG** and an **Assert** class is so common that .NET has built-in support for just this behavior. If you compile with DEBUG defined, either by putting **#define DEBUG** in your code or by compiling with **csc /d:DEBUG,** the Microsoft enables the **Debug** class in the **System.Diagnostics** namespace, which includes an **Assert** method. The

Debug.Assert( ) raises a dialog box, which is fine for manual debugging, but not very helpful for automated testing. Additionally, the **Debug** class and a companion **Trace** class (enabled by defining **TRACE**) have methods that output strings to a set of **TraceListener**s. **TraceListener**s can be used to send data to the system's event logs, the console, or custom sinks. We'll cover the use of these classes in Chapter 12.

C#'s access specifiers

When used, the C# access specifiers **public**, **internal, protected, protected internal** and **private** are placed in front of each definition for each member in your class, whether it's a field, method, or property. Each access specifier controls the access for only that particular definition. This is a distinct contrast to C++, in which the access specifier controls all the definitions following it until another access specifier comes along.

One way or another, everything has some kind of access specified for it. Even when not specified, each program component has a default access:

Element	Default Access
enum and **interface**	**public**
Non-nested **Class** and **struct**	**internal**
All type members (methods, properties, fields, etc.)	**private**

In the following sections, you'll learn all about the various types of access, starting with the default access.

public**: interface access**

When you use the **public** keyword, it means that the member declaration that immediately follows **public** is available to everyone, in particular to the client programmer who uses the library. Suppose you define a namespace **dessert** containing the following compilation unit:

```
//:c06:dessert:Cookie.cs
//Compile with
//csc /target:library Cookie.cs /out:dessert.dll
// Creates a library.
using System;
```

```
namespace Dessert{
   public class Cookie {
     public Cookie() {
       Console.WriteLine("Cookie constructor");
     }
     void Bite() { Console.WriteLine("Bite"); }
   }
}///:~
```

Now if you create a program that uses **Cookie**:

```
//:c06:Dinner.cs
//Compile with
//csc /reference:Dessert.dll Dinner.cs
// Uses the library.
using Dessert;
using System;

public class Dinner {
  public Dinner() {
   Console.WriteLine("Dinner constructor");
  }
  public static void Main() {
    Cookie x = new Cookie();
    //! x.Bite(); // Can't access
  }
}///:~
```

you can create a **Cookie** object, since its constructor is **public** and the class is **public**. (We'll look more at the concept of a **public** class later.) However, the **Bite( )** member is inaccessible inside **Dinner.cs** since **Bite( )** is **private**.

internal

What if you give no access specifier at all, as in all the examples before this chapter? The default access for a type is **internal**, which is sometimes referred to as "friendly." It means that all the other classes in the current assembly have access to the **internal** member, but to all the classes outside of this assembly the member appears to be **private**.

Internal access allows you to group related classes together in an assembly so that they can easily create each other. When you put classes together in an assembly you "own" the code in that package. It makes sense that only code you own should have **internal** access to other code you own. You could say that

internal access gives a meaning or a reason for grouping classes together in an assembly. In many languages the way you organize your definitions in files can be willy-nilly, but in C# you're compelled to organize them in a sensible fashion. In addition, you'll probably want to exclude classes that shouldn't have access to the classes being defined in the current assembly.

The class controls which code has access to its members. There's no magic way to "break in." Code from another assembly can't show up and say, "Hi, I'm a friend of **Bob**'s!" and expect to see the **protected**, **internal**, **protected internal**, and **private** members of **Bob**. The only way to grant access to a member is to:

1. Make the member **public**. Then everybody, everywhere, can access it.
2. Make the member **internal** by leaving off any access specifier if it's a class or by adding the **internal** keyword if it's a method or property, and put the other classes in the same assembly. Then the other classes can access the member.
3. As you'll see in Chapter 7, when inheritance is introduced, an inherited class can access a **protected** member as well as a **public** member (but not **private** members). It can access **internal** members only if the two classes are in the same assembly. But don't worry about that now. The same goes for **protected internal**, which allows access from an inherited member **or** from other classes in the assembly.
4. Expose the member via **public** properties that read and change the value. This is the most civilized approach in terms of OOP, and it is fundamental to C#, as you'll see in Chapter 7.

private**: you can't touch that!**

The **private** keyword means that no one can access that member except that particular class, inside methods of that class. Other classes in the same assembly cannot access **private** members, so it's as if you're even insulating the class against yourself. On the other hand, it's not unlikely that an assembly might be created by several people collaborating, so **private** allows you to freely change that member without concern that it will affect another class in the same assembly. The default access type for the internals of a type is **private**.

The default **internal** access for types and **private** for type members generally provide an adequate amount of hiding; remember, an **internal** member is inaccessible to the user of the assembly. This is nice, since the default access is the normal amount of caution you will use (and the one that you'll get if you forget to add any access control). Thus, you'll typically think about access for the

members that you explicitly want to make **public** for the client programmer, and as a result, you might not initially think you'll use the **private** keyword often since it's tolerable to get away without it. (This is a distinct contrast with C++.) However, it turns out that the consistent use of **private** is very important, especially where multithreading is concerned. (As you'll see in Chapter 16.)

Here's an example of the use of **private**:

```
//:c06:IceCream.cs
// Demonstrates "private" keyword.
using System;

class Sundae {
  private Sundae() {
    Console.WriteLine("private methods cannot be called"
    + " from methods not defined in class");
  }
  static internal Sundae MakeSundae() {
    Console.WriteLine("Sundae.MakeSundae() calls private");
    return new Sundae();
  }
}

public class IceCream {
  public static void Main() {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.MakeSundae();
  }
} ///:~
```

This shows an example in which **private** comes in handy: you might want to control how an object is created and prevent someone from directly accessing a particular constructor (or all of them). In the example above, you cannot create a **Sundae** object via its constructor; instead you must call the **MakeSundae( )** method to do it for you.

Any method that you're certain is only a "helper" method for that class should be kept **private,** to ensure that you don't accidentally use it elsewhere in the package and thus prohibit yourself from changing or removing the method. Keeping a method **private** guarantees that you retain this option.

The same is true for a **private** field inside a class. Unless you must expose the underlying implementation (which is a much rarer situation than you might

think), you should keep all fields **private**. However, just because a reference to an object is **private** inside a class doesn't mean that some other object can't have a **public** reference to the same object via other routes; if a **public** property returns a reference to a **private** object, the client can manipulate it freely.

protected

The **protected** access specifier requires a jump ahead to understand. First, you should be aware that you don't need to understand this section to continue through this book up through inheritance (Chapter 7). But for completeness, here is a brief description and example using **protected**.

The **protected** keyword deals with a concept called *inheritance*, which takes an existing class and adds new members to that class without touching the existing class, which we refer to as the *base class*. You can also change the behavior of existing members of the class. To inherit from an existing class, you add a colon and the name of the class from which it inherits, like this:

```
class Foo : Bar {
```

The rest of the class definition looks the same.

If you create a new assembly and you inherit from a class in another assembly, the only members you have access to are the **public** members of the original assembly. (Of course, if you perform the inheritance in the *same* assembly, you have the normal namespace access to all the **internal** members.) Sometimes the creator of the base class would like to take a particular member and grant access to derived classes but not the world in general. That's what **protected** does. If you refer back to the file **Cookie.cs**, the following class *cannot* access the **internal** member:

```
//:c06:ChocolateChip.cs
//Compile with:
// csc /reference:Dessert.dll ChocolateChip.cs
// Can't access internal member
// in parent class in another namespace.
using System;
using Dessert;

public class ChocolateChip : Cookie {
  public ChocolateChip() {
    Console.WriteLine("ChocolateChip constructor");
  }
  public static void Main() {
```

```
    ChocolateChip x = new ChocolateChip();
    //! x.Bite(); // Still can't access bite
  }
} //:~
```

One of the interesting things about inheritance is that if a method **Bite( )** exists in class **Cookie**, then it also exists in any class inherited from **Cookie**. But since **Bite( )** is **internal** to a foreign assembly, it's unavailable to us in this one. Of course, you could make **Bite( ) public**, but then everyone would have access and maybe that's not what you want. If we change the class **Cookie** as follows:

```
public class Cookie {
  public Cookie() {
    Console.WriteLine("Cookie constructor");
  }
  protected void Bite() {
    Console.WriteLine("bite");
  }
}
```

then **Bite( )** no longer provides **internal** access within the **Dessert** assembly, but it is now accessible to anyone inheriting from **Cookie**. However, it is *not* **public**. If you wish to keep **Bite( )** so that it still has internal access within the **Dessert** assembly and also have it accessible from inherited classes, you can declare it **protected internal**.

Interface and implementation

Access control is often referred to as *implementation hiding*. Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation*[1]. The result is a data type with characteristics and behaviors.

Access control puts boundaries within a data type for two important reasons. The first is to establish what the client programmers can and can't use. You can build your internal mechanisms into the structure without worrying that the client programmers will accidentally treat the internals as part of the interface that they should be using.

This feeds directly into the second reason, which is to separate the interface from the implementation. If the structure is used in a set of programs, but client

[1] However, other people refer to implementation hiding alone as encapsulation.

programmers can't do anything but send messages to the **public** interface, then you can change anything that's *not* **public** (e.g., **internal**, **protected**, **protected internal**, or **private**) without requiring modifications to client code.

We're now in the world of object-oriented programming, where a class is actually describing "a class of objects," as you would describe a class of fishes or a class of birds. Any object belonging to this class will share these characteristics and behaviors. The class is a description of the way all objects of this type will look and act.

In the original OOP language, Simula-67, the keyword **class** was used to describe a new data type. The same keyword has been used for most object-oriented languages. This is the focal point of the whole language: the creation of new data types that are more than just boxes containing data and methods.

The class is the fundamental OOP concept.

For clarity, you might prefer a style of creating classes that puts the **public** members at the beginning, followed by the **protected**, **internal**, and **private** members. The advantage is that the user of the class can then read down from the top and see first what's important to them (the **public** members, because they can be accessed outside the file), and stop reading when they encounter the non-**public** members, which are part of the internal implementation:

```
public class X {
  public void Pub1( ) { /* . . . */ }
  public void Pub2( ) { /* . . . */ }
  public void Pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

This will make it only partially easier to read because the interface and implementation are still mixed together. That is, you still see the source code—the implementation—because it's right there in the class. In addition, the comment documentation somewhat lessens the importance of code readability by the client programmer. Displaying the interface to the consumer of a class is really the job of the *class browser*, a tool whose job is to look at all the available classes and show you what you can do with them (i.e., what members are

available) in a useful fashion. Microsoft's Visual Studio .NET is the first, but not the only, tool to provide a class browser for C#.

Class access

In C#, the access specifiers can also be used to determine which classes *within* a library will be available to the users of that library. If you want a class to be available to a client programmer, you place the **public** keyword somewhere before the opening brace of the class body. This controls whether the client programmer can even create an object of the class.

To control the access of a class, the specifier must appear before the keyword **class**. Thus you can say:

```
public class Widget {
```

Now if the name of your library is **Mylib** any client programmer can access **Widget** by saying

```
using Mylib;
Widget w;
```

What if you've got a class inside **Mylib** that you're just using to accomplish the tasks performed by **Widget** or some other **public** class in **Mylib**? You don't want to go to the bother of creating documentation for the client programmer, and you think that sometime later you might want to completely change things and rip out your class altogether, substituting a different one. To give you this flexibility, you need to ensure that no client programmers become dependent on your particular implementation details hidden inside **Mylib**. To accomplish this, you just leave the **public** keyword off the class, in which case it becomes **internal**. (That class can be used only within that assembly.)

Note that a class cannot be **private** (that would make it accessible to no one but the class), or **protected**. So you have only two choices for class access: **internal** or **public**. If you don't want anyone else to have access to that class, you can make all the constructors **private**, thereby preventing anyone but you, inside a **static** member of the class, from creating an object of that class[2]. Here's an example:

```
//:c06:Lunch.cs
```

[2] You can also do it by inheriting (Chapter 7) from that class.

```
// Demonstrates class access specifiers.
// Make a class effectively private
// with private constructors:

class Soup {
  private Soup() {}
  // (1) Allow creation via static method:
  public static Soup MakeSoup() {
    return new Soup();
  }
  // (2) Create a static object and
  // return a reference upon request.
  // (The "Singleton" pattern):
  private static Soup ps1 = new Soup();
  public static Soup Access() {
    return ps1;
  }
  public void F() {}
}

class Sandwich { // Uses Lunch
  void F() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
  void Test() {
    // Can't do this! Private constructor:
    //!Soup priv1 = new Soup();
    Soup priv2 = Soup.MakeSoup();
    Sandwich f1 = new Sandwich();
    Soup.Access().F();
  }
} ///:~
```

Up to now, most of the methods have been returning either **void** or a primitive type, so the definition:

```
  public static Soup Access() {
    return ps1;
  }
```

might look a little confusing at first. The word before the method name (**Access**) tells what the method returns. So far this has most often been **void,** which means it returns nothing. But you can also return a reference to an object, which is what happens here. This method returns a reference to an object of class **Soup**.

The class **Soup** shows how to prevent direct creation of a class by making all the constructors **private**. Remember that if you don't explicitly create at least one constructor, the default constructor (a constructor with no arguments) will be created for you. By writing the default constructor, it won't be created automatically. By making it **private**, no one can create an object of that class. But now how does anyone use this class? The above example shows two options. First, a **static** method is created that creates a new **Soup** and returns a reference to it. This could be useful if you want to do some extra operations on the **Soup** before returning it, or if you want to keep count of how many **Soup** objects to create (perhaps to restrict their population).

The second option uses what's called a *design pattern*, which is covered in *Thinking in Patterns with Java*, downloadable at *www.BruceEckel.com*. This particular pattern is called a *Singleton* because it allows only a single object to ever be created. The object of class **Soup** is created as a **static private** member of **Soup**, so there's one and only one, and you can't get at it except through the **public** method **Aaccess( )**.

As previously mentioned, if you don't put an access specifier for class access it defaults to **internal**. This means that an object of that class can be created by any other class in the assembly, but not outside the assembly. However, if a **static** member of that class is **public**, the client programmer can still access that **static** member even though they cannot create an object of that class.

Summary

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the user of that library—the client programmer—who is another programmer, but one putting together an application or using your library to build a bigger library.

Without rules, client programmers can do anything they want with all the members of a class, even if you might prefer they don't directly manipulate some of the members. Everything's naked to the world.

This chapter looked at how classes are built to form libraries: first, the way a group of classes is packaged within a library, and second, the way the class controls access to its members.

It is estimated that a C programming project begins to break down somewhere between 50K and 100K lines of code because C has a single "name space" so names begin to collide, causing an extra management overhead. In C#, the **namespace** keyword, the assembly referencing scheme, and the **using** keyword give you complete control over names, so the issue of name collision is easily avoided.

There are two reasons for controlling access to members. The first is to keep users' hands off tools that they shouldn't touch: tools that are necessary for the internal machinations of the data type, but not part of the interface that users need to solve their particular problems. So making methods and fields **private** is a service to users because they can easily see what's important to them and what they can ignore. It simplifies their understanding of the class.

The second and most important reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. You might build a class one way at first, and then discover that restructuring your code will provide much greater speed. If the interface and implementation are clearly separated and protected, you can accomplish this without forcing the users to rewrite their code.

Access specifiers in C# give valuable control to the creator of a class. The users of the class can clearly see exactly what they can use and what to ignore. More important, though, is the ability to ensure that no user becomes dependent on any part of the underlying implementation of a class. If you know this as the creator of the class, you can change the underlying implementation with the knowledge that no client programmer will be affected by the changes because they can't access that part of the class.

When you have the ability to change the underlying implementation, you can not only improve your design later, but you also have the freedom to make mistakes. No matter how carefully you plan and design you'll make mistakes. Knowing that it's relatively safe to make these mistakes means you'll be more experimental, you'll learn faster, and you'll finish your project sooner.

The public interface to a class is what the user *does* see, so that is the most important part of the class to get "right" during analysis and design. Even that allows you some leeway for change. If you don't get the interface right the first time, you can *add* more methods, as long as you don't remove any that client programmers have already used in their code.

Exercises

1. Write a program that creates an **ArrayList** object without explicitly importing **System.Collections**.

2. In the section labeled "the library unit," turn the code fragments concerning **MyNamespace** into a compiling and running pair of assemblies (a library assembly and an executable assembly).

3. In the section labeled "Collisions," take the code fragments and turn them into a program, and verify that collisions do in fact occur.

4. Compile **TestDebug.cs** with **Debug** defined and not. Confirm that the behavior is different due to the preprocessed directives.

5. Create a class with **public**, **private**, **protected,** and **internal** data members and method members. Create an object of this class and see what kind of compiler messages you get when you try to access all the class members. Be aware that classes in the same assembly are part of the "default" package.

6. Create a class with **internal** properties and methods. Compile it into a library assembly. Create another class that attempts to read the data and compile it into an executable assembly. Observe the behavior. Compile both classes into a single executable assembly and observe the behavior.

7. Change the class **Cookie** as specified in the section labeled "**protected**: 'sort of friendly.'" Verify that **Bite( )** is not **public**.

8. In the section titled "Class access" you'll find code fragments describing **Mylib** and **Widget**. Create this library, then create a **Widget** in a class that is not part of the **Mylib** package.

9. Following the form of the example **Lunch.cs**, create a class called **ConnectionManager** that manages a fixed array of **Connection** objects. The client programmer must not be able to explicitly create **Connection** objects, but can only get them via a **static** method in **ConnectionManager**. When the **ConnectionManager** runs out of objects, it returns a **null** reference. Test the classes in **Main( )**.

10. Referring back to the party domain exercises from earlier chapters, divide the domain into several logical namespaces.

11. Divide a large sheet of paper with the namespaces you developed in exercise 10. Place the classes that you have developed into their appropriate namespace. Using the programs that you have written, trace execution with a coin.

    If the coin accesses a method or property in the same class, mark the method or property as **private**.

    If the coin accesses method or property in a class within its namespace, mark the method or data as **internal**.

    When the coin crosses namespace boundaries, it necessarily must be accessing something **public**.

    Modify your code according to this exercise. Compile each namespace into a separate assembly. Confirm that all your programs still work!

12. Taking the diagram developed in the previous exercise as a start, create a new diagram that lists just the public classes, methods, and properties in your party domain. Compare the readability of this diagram with the complete diagram.

13. With the diagrams from the two previous exercises available, implement a program in one of the namespaces that you haven't concentrated on. You should find that you need to access classes in other namespaces. Eliminate common code and confirm that all your programs continue to work. Are the diagrams helpful? Is one more helpful than the other? Why?

7: Reusing Classes

One of the most compelling features about object orientation is code reuse. Studies have shown that high-quality reusable components can be the second-most important factor to productivity (only the order-of-magnitude difference in productivity between the best and worst programmers counts more). Conversely, trying to work with low-quality reusable components is the greatest detractor from productivity. In other words, the quality of the libraries that you use to build your solutions has an enormous influence on software success.

Like everything in C#, the key to software quality lies in the object-oriented class. You reuse code by creating new classes, but instead of creating them from scratch, you use existing classes that someone has already built and debugged.

The trick is to use the classes without soiling the existing code. In this chapter you'll see two ways to accomplish this. The first is quite straightforward: You simply create objects of your existing class inside the new class. This is called *composition,* because the new class is composed of objects of existing classes. You're simply reusing the functionality of the code, not its form.

The second approach is more subtle. It creates a new class as a *type of* an existing class. You literally take the form of the existing class and add code to it without modifying the existing class. This magical act is called *inheritance*, and the compiler does most of the work. Inheritance is one of the cornerstones of object-oriented programming.

It turns out that much of the syntax and behavior are similar for both composition and inheritance (which makes sense because they are both ways of making new types from existing types). In this chapter, you'll learn about these code reuse mechanisms.

Composition syntax

Until now, composition has been used quite frequently. You simply place object references inside new classes. For example, suppose you'd like an object that holds several **string** objects, a couple of primitives, and an object of another class. For the non-value types, you put references inside your new class, but you define the objects directly:

```
//:c07:SprinklerSystem.cs
// Composition for code reuse.
using System;

class WaterSource {
  private string s;
  internal WaterSource() {
    Console.WriteLine("WaterSource()");
    s = "Constructed";
  }
  public override string ToString() { return s;}
}

public class SprinklerSystem {
  private string valve1, valve2, valve3, valve4;
  WaterSource source;
  int i;
  float f;
  void Print() {
    Console.WriteLine("valve1 = " + valve1);
    Console.WriteLine("valve2 = " + valve2);
    Console.WriteLine("valve3 = " + valve3);
    Console.WriteLine("valve4 = " + valve4);
    Console.WriteLine("i = " + i);
    Console.WriteLine("f = " + f);
    Console.WriteLine("source = " + source);
  }
  public static void Main() {
    SprinklerSystem x = new SprinklerSystem();
    x.Print();
  }
} ///:~
```

At first glance, you might assume—C# being as safe and careful as it is—that the compiler would automatically construct objects for each of the references in the above code; for example, calling the default constructor for **WaterSource** to initialize **source**. The output of the print statement is in fact:

```
valve1 =
valve2 =
valve3 =
valve4 =
i = 0
f = 0
source =
```

Value types that are fields in a class are automatically initialized to zero, as noted in Chapter 5. But the object references are initialized to **null**, and if you try to call methods for any of them you'll get an exception. It's actually pretty good (and useful) that you can still print them out without throwing an exception.

It makes sense that the compiler doesn't just create a default object for every reference because that would incur unnecessary overhead in many cases. If you want the references initialized, you can do it:

1. At the point the objects are defined. This means that they'll always be initialized before the constructor is called.
2. In the constructor for that class.
3. Right before you actually need to use the object. This is often called *lazy initialization*. It can reduce overhead in situations where the object doesn't need to be created every time.

All three approaches are shown here:

```
//:c07:Bath.cs
// Constructor initialization with composition.
using System;

class Soap {
  private string s;
  internal Soap() {
    Console.WriteLine("Soap()");
    s = "Constructed";
  }
  public override string ToString() { return s;}
```

```
}

public class Bath {
  private string
  // Initializing at point of definition:
  s1 = "Happy",
  s2 = "Happy",
  s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    Console.WriteLine("Inside Bath()");
// Initializing inside the constructor
    s3 = "Joy";
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void Print() {
    // Delayed initialization:
    if (s4 == null)
      s4 = "Joy";
    Console.WriteLine("s1 = " + s1);
    Console.WriteLine("s2 = " + s2);
    Console.WriteLine("s3 = " + s3);
    Console.WriteLine("s4 = " + s4);
    Console.WriteLine("i = " + i);
    Console.WriteLine("toy = " + toy);
    Console.WriteLine("castille = " + castille);
  }
  public static void Main() {
    Bath b = new Bath();
    b.Print();
  }
} ///:~
```

Note that in the **Bath** constructor a statement is executed before any of the initializations take place. When you don't initialize at the point of definition, there's still no guarantee that you'll perform any initialization before you send a message to an object reference—except for the inevitable run-time exception.

Here's the output for the program:

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

When **Print( )** is called it fills in **s4** so that all the fields are properly initialized by the time they are used.

Inheritance syntax

Inheritance is an integral part of C# (and OOP languages in general). It turns out that you're always doing inheritance when you create a class, because unless you explicitly inherit from some other class, you implicitly inherit from C#'s standard root class **object**.

The syntax for composition is obvious, but to perform inheritance there's a distinctly different form. When you inherit, you say "This new class is like that old class." You state this in code by giving the name of the class as usual, but before the opening brace of the class body, put a colon followed by the name of the *base class*. When you do this, you automatically get all the data members and methods in the base class. Here's an example:

```
//:c07:Detergent.cs
///Compile with: "/main:Detergent"
// Inheritance syntax & properties.
using System;

internal class Cleanser {
  private string s = "Cleanser";
  public void Append(string a) { s += a;}
  public void Dilute() { Append(" dilute()");}
  public void Apply() { Append(" apply()");}
  virtual public void Scrub() { Append(" scrub()");}
  public void Print() { Console.WriteLine(s);}
  public static void Main() {
    Cleanser x = new Cleanser();
```

```
    x.Dilute(); x.Apply(); x.Scrub();
    x.Print();
  }
}

internal class Detergent : Cleanser {
  // Change a method:
  override public void Scrub() {
    Append(" Detergent.scrub()");
    base.Scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void Foam() { Append(" Foam()");}
  // Test the new class:
  new public static void Main() {
    Detergent x = new Detergent();
    x.Dilute();
    x.Apply();
    x.Scrub();
    x.Foam();
    x.Print();
    Console.WriteLine("Testing base class:");
    Cleanser.Main();
  }
} ///:~
```

This demonstrates a number of features. First, both **Cleanser** and **Detergent** contain a **Main( )** method. You can create a **Main( )** for each one of your classes, but if you do so, the compiler will generate an error, saying that you are defining multiple entry points. You can choose which **Main( )** you want to have associated with the assembly by using the /Main:Classname switch. Thus, if you compile the above with **csc Detergent.cs /Main:Cleanser**, the output will be:

```
Cleanser dilute() apply() scrub()
```

While if compiled with **csc Detergent.cs /Main:Detergent**, the result is:

```
Cleanser dilute() apply() Detergent.scrub() scrub() Foam()
Testing base class:
Cleanser dilute() apply() scrub()
```

This technique of putting a **Main( )** in each class can sometimes help with testing, when you just want to write a quick little program to make sure your methods are working the way you intend them to. But for general testing

purposes, you should use a unit-testing framework (see Appendix C). You don't need to remove the **Main( )** when you're finished testing; you can leave it in for later testing.

Here, you can see that **Detergent.Main( )** calls **Cleanser.Main( )** explicitly, passing it the same arguments from the command line (however, you could pass it any **string** array).

It's important that all of the methods in **Cleanser** are **public**. Remember that if you leave off any access specifier the member defaults to **private**, which allows access only to the very class in which the field or method is defined. So to plan for inheritance, as a general rule leave fields **private**, but make all methods **public**. (**protected** members also allow access by derived classes; you'll learn details on what this means later.) Of course, in particular cases you must make adjustments, but this is a useful guideline.

Note that **Cleanser** has a set of methods in its interface: **Append( )**, **Dilute( )**, **Apply( )**, **Scrub( )**, and **Print( )**. Because **Detergent** is *derived from* **Cleanser** it automatically gets all these methods in its interface, even though you don't see them all explicitly defined in **Detergent**. You can think of inheritance, then, as *reusing the interface*. (The implementation also comes with it, but that part isn't the primary point.)

As seen in **Scrub( )**, it's possible to take a method that's been defined in the base class and modify it. In this case, you might want to call the method from the base class inside the new version. But inside **Scrub( )** you cannot simply call **Scrub( )**, since that would produce a recursive call, which isn't what you want. To solve this problem C# has the keyword **base** that refers to the "base class" (also called the "superclass") from which the current class has been inherited. Thus the expression **base.Scrub( )** calls the base-class version of the method **Scrub( )**.

When inheriting you're not restricted to using the methods of the base class. You can also add new methods to the derived class exactly the way you put any method in a class: just define it. The method **Foam( )** is an example of this.

In **Detergent.Main( )** you can see that for a **Detergent** object you can call all the methods that are available in **Cleanser** as well as in **Detergent** (i.e., **Foam( )**).

Initializing the base class

Since there are now two classes involved—the base class and the derived class—instead of just one, it can be a bit confusing to try to imagine the resulting object

produced by a derived class. From the outside, it looks like the new class has the same interface as the base class and maybe some additional methods and fields. But inheritance doesn't just copy the interface of the base class. When you create an object of the derived class, it contains within it a *subobject* of the base class. This subobject is the same as if you had created an object of the base class by itself. It's just that, from the outside, the subobject of the base class is wrapped within the derived-class object.

Of course, it's essential that the base-class subobject be initialized correctly and there's only one way to guarantee that: perform the initialization in the constructor, by calling the base-class constructor, which has all the appropriate knowledge and privileges to perform the base-class initialization. C# automatically inserts calls to the base-class constructor in the derived-class constructor. The following example shows this working with three levels of inheritance:

```
//:c07:Cartoon.cs
// Constructor calls during inheritance.
using System;

internal class Art {
  protected Art() {
    Console.WriteLine("Art constructor");
  }
}

internal class Drawing : Art {
  protected Drawing() {
    Console.WriteLine("Drawing constructor");
  }
}

internal class Cartoon : Drawing {
  protected Cartoon() {
    Console.WriteLine("Cartoon constructor");
  }
  public static void Main() {
    Cartoon x = new Cartoon();
  }
} ///:~
```

The output for this program shows the automatic calls:

```
Art constructor
Drawing constructor
Cartoon constructor
```

You can see that the construction happens from the base "outward," so the base class is initialized before the derived-class constructors can access it.

Even if you don't create a constructor for **Cartoon( )**, the compiler will synthesize a default constructor for you that calls the base class constructor.

Constructors with arguments

The above example has default constructors; that is, they don't have any arguments. It's easy for the compiler to call these because there's no question about what arguments to pass. If your class doesn't have default arguments, or if you want to call a base-class constructor that has an argument, you must explicitly write the calls to the base-class constructor using the **base** keyword and the appropriate argument list:

```
//:c07:Chess.cs
// Inheritance, constructors and arguments.
using System;

public class Game {
  internal Game(int i) {
    Console.WriteLine("Game constructor");
  }
}

public class BoardGame : Game {
  internal BoardGame(int i) : base(i) {
    Console.WriteLine("BoardGame constructor");
  }
}

public class Chess : BoardGame {
  internal Chess() : base(11){
    Console.WriteLine("Chess constructor");
  }
  public static void Main() {
    Chess x = new Chess();
  }
}///:~
```

If you don't call the base-class constructor in **BoardGame( )**, the compiler will complain that it can't find a constructor of the form **Game( )**.

Catching base constructor exceptions

As just noted, the compiler forces you to place the base-class constructor call before even the body of the derived-class constructor. As you'll see in Chapter 11, this also prevents a derived-class constructor from catching any exceptions that come from a base class. This can be inconvenient at times.

```
//:c07:Dome.cs
using System;

class Dome {
  public Dome(){
    throw new InvalidOperationException();
  }
}

class Brunelleschi : Dome {
  public Brunelleschi(){
    Console.WriteLine("Ingenious Vaulting");
  }

  public static void Main(){
    try {
      new Brunelleschi();
    } catch (Exception ex) {
      Console.WriteLine(ex);
    }
  }
}///:~
```

prints:

```
System.InvalidOperationException: Operation is not valid
due to the current state of the object.
   at Dome..ctor()
   at Brunelleschi.Main()
```

Combining composition and inheritance

It is very common to use composition and inheritance together. The following example shows the creation of a more complex class, using both inheritance and composition, along with the necessary constructor initialization:

```
//:c07:PlaceSetting.cs
// Combining composition & inheritance.
using System;

class Plate {
  internal Plate(int i) {
    Console.WriteLine("Plate constructor");
  }
}

class DinnerPlate : Plate {
  internal DinnerPlate(int i) : base(i) {
    Console.WriteLine("DinnerPlate constructor");
  }
}

class Utensil {
  internal Utensil(int i) {
    Console.WriteLine("Utensil constructor");
  }
}

class Spoon : Utensil {
  internal Spoon(int i) : base(i) {
    Console.WriteLine("Spoon constructor");
  }
}

class Fork : Utensil {
  internal Fork(int i) : base(i) {
    Console.WriteLine("Fork constructor");
  }
}
```

```
class Knife : Utensil {
  internal Knife(int i) : base(i) {
    Console.WriteLine("Knife constructor");
  }
}

// A cultural way of doing something:
class Custom {
  internal Custom(int i) {
    Console.WriteLine("Custom constructor");
  }
}

class PlaceSetting : Custom {
  Spoon sp;
  Fork frk;
  Knife kn;
  DinnerPlate pl;
  PlaceSetting(int i) : base(i + 1) {
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    Console.WriteLine("PlaceSetting constructor");
  }
  public static void Main (){
    PlaceSetting x = new PlaceSetting(9);
  }
}///:~
```

While the compiler forces you to initialize the base classes, and requires that you do it right at the beginning of the constructor, it doesn't watch over you to make sure that you initialize the member objects, so you must remember to pay attention to that.

Guaranteeing proper cleanup

You may recall from Chapter 5 that although C# has a destructor, we said that the proper way to guarantee that an object cleans up after itself involved the **IDisposable** interface, implementing the method **Dispose( ),** and wrapping the "valuable resource" in a **using** block. At the time, we deferred a discussion of how it worked, but with an understanding of inheritance, it starts to becomes

clear. (Although understanding how the **using** block works will require an understanding of Exceptions, which is coming in Chapter 11.)

Consider an example of a computer-aided design system that draws pictures on the screen:

```
//:c07:CADSystem.cs
// Ensuring proper cleanup.
using System;

class Shape : IDisposable {
  internal Shape(int i) {
    Console.WriteLine("Shape constructor");
  }
  public virtual void Dispose() {
    Console.WriteLine("Shape disposed");
  }
}

class Circle : Shape {
  internal Circle(int i) : base(i) {
    Console.WriteLine("Drawing a Circle");
  }
  public override void Dispose() {
    Console.WriteLine("Erasing a Circle");
    base.Dispose();
  }
}

class Triangle : Shape {
  internal Triangle(int i) : base(i) {
    Console.WriteLine("Drawing a Triangle");
  }
  public override void Dispose() {
    Console.WriteLine("Erasing a Triangle");
    base.Dispose();
  }
}

class Line : Shape {
  private int start, end;
  internal Line(int start, int end) : base(start){
```

```
    this.start = start;
    this.end = end;
    Console.WriteLine("Drawing a Line: "
      + start + ", " + end);
  }
  public override void Dispose() {
    Console.WriteLine("Erasing a Line: "
      + start + ", " + end);
    base.Dispose();
  }
}

class CADSystem : Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[10];
  CADSystem(int i) : base(i + 1){
    for (int j = 0; j < 10; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    Console.WriteLine("Combined constructor");
  }
  public override void Dispose() {
    Console.WriteLine("CADSystem.Dispose()");
    // The order of cleanup is the reverse
    // of the order of initialization
    t.Dispose();
    c.Dispose();
    for (int i = lines.Length - 1; i >= 0; i--)
      lines[i].Dispose();
    base.Dispose();
  }
  public static void Main() {
    CADSystem x = new CADSystem(47);
    using(x){
      // Code and exception handling...
    }
    Console.WriteLine("Using block left");
  }
}///:~
```

Everything in this system is some kind of **Shape** (which itself is a kind of **object** since it's implicitly inherited from the root class and which implements an *interface* called IDisposable). Each class redefines **Shape**'s **Dispose( )** method in addition to calling the base-class version of that method using **base**. The specific **Shape** classes—**Circle**, **Triangle** and **Line**—all have constructors that "draw," although any method called during the lifetime of the object could be responsible for doing something that needs cleanup. Each class has its own **Dispose( )** method to restore nonmemory things back to the way they were before the object existed.

In **Main( )**, you can see the **using** keyword in action. A **using** block takes an **IDisposable** as an argument. When execution leaves the block (even if an exception is thrown), **IDisposable.Dispose( )** is called. But because we have implemented **Dispose( )** in **Shape** and all the classes derived from it, inheritance kicks in and the appropriate **Dispose( )** method is called. In this case, the using block has a **CADSystem**. Its **Dispose( )** method calls, in turn, the **Dispose( )** method of the objects which comprise it.

Note that in your cleanup method you must also pay attention to the calling order for the base-class and member-object cleanup methods in case one subobject depends on another. In general, you should follow the same form that is imposed by a C++ compiler on its destructors: First perform all of the cleanup work specific to your class, in the reverse order of creation. (In general, this requires that base-class elements still be viable.) Then call the base-class **Dispose** method, as demonstrated here.

There can be many cases in which the cleanup issue is not a problem; you just let the garbage collector do the work. But when you must do it explicitly, diligence and attention is required.

Order of garbage collection

There's not much you can rely on when it comes to garbage collection. The garbage collector may not be called until your program exits. If it is called, it can reclaim objects in any order it wants. It's best to not rely on garbage collection for anything but memory reclamation. If you have "valuable resources" which need explicit cleanup, always initialize them as late as possible, and dispose of them as soon as you can.

Choosing composition vs. inheritance

Both composition and inheritance allow you to place subobjects inside your new class. You might wonder about the difference between the two, and when to choose one over the other.

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object so that you can use it to implement functionality in your new class, but the user of your new class sees the interface you've defined for the new class rather than the interface from the embedded object. For this effect, you embed **private** objects of existing classes inside your new class.

Sometimes it makes sense to allow the class user to directly access the composition of your new class, that is, to make the member objects **public**. The member objects use implementation hiding themselves, so this is a safe thing to do. When the user knows you're assembling a bunch of parts, it makes the interface easier to understand. A **Car** object is a good example:

```
//:c07:Car.cs
// Composition with public objects.

public class Engine {
  public void Start() {}
  public void Rev() {}
  public void Stop() {}
}

public class Wheel {
  public void Inflate(int psi) {}
}

public class Window {
  public void Rollup() {}
  public void Rolldown() {}
}

public class Door {
  public Window window = new Window();
  public void Open() {}
```

```
  public void Close() {}
}

public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door left = new Door(),
    right = new Door();          // 2-door
  public Car() {
    for (int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void Main() {
    Car car = new Car();
    car.left.window.Rollup();
    car.wheel[0].Inflate(72);
  }
} ///:~
```

Because the composition of a car is part of the analysis of the problem (and not simply part of the underlying design), making the members **public** assists the client programmer's understanding of how to use the class and requires less code complexity for the creator of the class. However, keep in mind that this is a special case and that in general you should make fields **private**.

When you inherit, you take an existing class and make a special version of it. In general, this means that you're taking a general-purpose class and specializing it for a particular need. With a little thought, you'll see that it would make no sense to compose a car using a vehicle object—a car doesn't contain a vehicle, it *is* a vehicle. The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

protected

Now that you've been introduced to inheritance, the keyword **protected** finally has meaning. In an ideal world, **private** members would always be hard-and-fast **private**, but in real projects there are times when you want to make something hidden from the world at large and yet allow access for members of derived classes. The **protected** keyword is a nod to pragmatism. It says "This is **private** as far as the class user is concerned, but available to anyone who inherits from this class."

The best tack to take is to leave the data members **private**—you should always preserve your right to change the underlying implementation. You can then allow controlled access to inheritors of your class through **protected** methods:

```
//:c07:Orc.cs
// The protected keyword.

public class Villain {
  private int i;
  protected int Read() { return i;}
  protected void Set(int ii) { i = ii;}
  public Villain(int ii) { i = ii;}
  public int Value(int m) { return m*i;}
}

public class Orc : Villain {
  private int j;
  public Orc(int jj) :base(jj) {  j = jj;}
  public void Change(int x) { Set(x);}
} ///:~ (non-executable code snippet)
```

You can see that **Change( )** has access to **Set( )** because it's **protected**.

Incremental development

One of the advantages of inheritance is that it supports *incremental development* by allowing you to introduce new code without causing bugs in existing code. This also isolates new bugs inside the new code. By inheriting from an existing, functional class and adding data members and methods (and redefining existing methods), you leave the existing code—that someone else might still be using—untouched and unbugged. If a bug happens, you know that it's in your new code, which is much shorter and easier to read than if you had modified the body of existing code.

It's rather amazing how cleanly the classes are separated. You don't even need the source code for the methods in order to reuse the code. This is true for both inheritance and composition.

It's important to realize that program development is an incremental process, just like human learning. You can do as much analysis as you want, but you still won't know all the answers when you set out on a project. You'll have much more success—and more immediate feedback—if you start out to "grow" your project as

an organic, evolutionary creature, rather than constructing it all at once like a glass-box skyscraper.

Although inheritance for experimentation can be a useful technique, at some point after things stabilize you need to take a new look at your class hierarchy with an eye to collapsing it into a sensible structure. Remember that underneath it all, inheritance is meant to express a relationship that says "This new class is a *type of* that old class." Your program should not be concerned with pushing bits around, but instead with creating and manipulating objects of various types to express a model in the terms that come from the problem space.

Upcasting

The most important aspect of inheritance is not that it provides methods for the new class. It's the relationship expressed between the new class and the base class. This relationship can be summarized by saying "The new class *is a type of* the existing class."

This description is not just a fanciful way of explaining inheritance—it's supported directly by the language. As an example, consider a base class called **Instrument** that represents musical instruments, and a derived class called **Wind**. Because inheritance means that all of the methods in the base class are also available in the derived class, any message you can send to the base class can also be sent to the derived class. If the **Instrument** class has a **Play( )** method, so will **Wind** instruments. This means we can accurately say that a **Wind** object is also a type of **Instrument**. The following example shows how the compiler supports this notion:

```
//:c07:Wind.cs
// Inheritance & upcasting.

public class Instrument {
  public void play() {}
  static internal void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind : Instrument {
```

```
  public static void Main() {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
} ///:~
```

What's interesting in this example is the **Tune( )** method, which accepts an **Instrument** reference. However, in **Wind**.**Main( )** the **Tune( )** method is called by giving it a **Wind** reference. Given that C# is particular about type checking, it seems strange that a method that accepts one type will readily accept another type, until you realize that a **Wind** object is also an **Instrument** object, and there's no method that **Tune( )** could call for an **Instrument** that isn't also in **Wind**. Inside **Tune( )**, the code works for **Instrument** and anything derived from **Instrument**, and the act of converting a **Wind** reference into an **Instrument** reference is called *upcasting*.

Why "upcasting"?

The reason for the term is historical, and based on the way class inheritance diagrams have traditionally been drawn: with the root at the top of the page, growing downward. (Of course, you can draw your diagrams any way you find helpful.) The inheritance diagram for **Wind.java** is then:

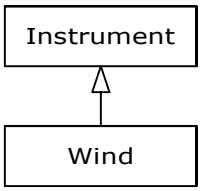


Figure 7-1: Traditionally, base classes are drawn higher on the page.

Casting from derived to base moves *up* on the inheritance diagram, so it's commonly referred to as *upcasting*. Upcasting is always safe because you're going from a more specific type to a more general type. That is, the derived class is a superset of the base class. It might contain more methods than the base class, but it must contain *at least* the methods in the base class. The only thing that can occur to the class interface during the upcast is that it can lose methods, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

You can also perform the reverse of upcasting, called *downcasting*, but this involves a dilemma that is the subject of Chapter 12.

Composition vs. inheritance revisited

In object-oriented programming, the most likely way that you'll create and use code is by simply packaging data and methods together into a class, and using objects of that class. You'll also use existing classes to build new classes with composition. Less frequently, you'll use inheritance. So although inheritance gets a lot of emphasis while learning OOP, it doesn't mean that you should use it everywhere you possibly can. On the contrary, you should use it sparingly, only when it's clear that inheritance is useful. One of the clearest ways to determine whether you should use composition or inheritance is to ask whether you'll ever need to upcast from your new class to the base class. If you must upcast, then inheritance is necessary, but if you don't need to upcast, then you should look closely at whether you need inheritance. The next chapter (polymorphism) provides one of the most compelling reasons for upcasting, but if you remember to ask "Do I need to upcast?" you'll have a good tool for deciding between composition and inheritance.

Explicit overloading only

Some of C#'s most notable departures from the object-oriented norm are the barriers it places on the road to overloading functionality. In most object-oriented languages, if you have classes Fork and Spoon that descend from Utensil, a base method **GetFood**, and two implementations of it, you just declare the method in the base and have identical signatures in the descending classes:

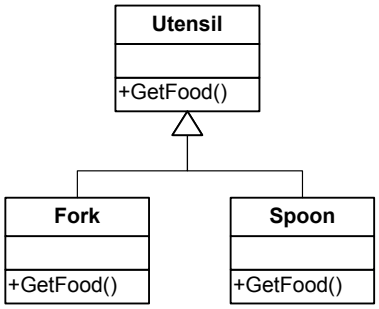


Figure 7-2: Fork and Spoon overload Utensil.GetFood( )

In Java, this would look like:

```
class Utensil{
   public void GetFood(){ //…}
}
```

```
class Fork extends Utensil{
   public void GetFood(){
      System.out.println("Spear");
   }
}

class Spoon extends Utensil{
   public void GetFood(){
      System.out.println("Scoop");
   }
}
```

In C#, you have to jump through a bit of a hoop; methods for which overloading is intended must be declared **virtual** and the overloading method must be declared as an **override**. To get the desired structure would look like this:

```
class Utensil{
   public virtual void GetFood(){ //…}
}

class Fork extends Utensil{
public override void GetFood(){
     Console.WriteLine("Spear");
   }
}

class Spoon extends Utensil{
public override void GetFood(){
      Console.WriteLine("Scoop");
   }
}
```

This is a behavior that stems from Microsoft's experience with "DLL Hell" and thoughts about a world in which object-oriented components are the building blocks of very large systems. Imagine that you are using Java and using a third-party "Kitchen" component that includes the base class of Utensil, but you customize it to use that staple of dorm life – the Spork. But in addition to implementing **GetFood( )**, you add a dorm-like method **Wash( )**:

```
//Spork.java
class Spork extends Utensil{
```

```
   public void GetFood(){
      System.out.println("Spear OR Scoop!");
   }

   public void Wash(){
      System.out.println("Wipe with napkin");
   }
}
```

Of course, since **Wash** isn't implemented in **Utensil**, you could only "wash" a spork (which is just as well, considering the unhygienic nature of the implementation). So the problem happens when the 3rd-party Kitchen component vendor releases a new version of their component, and this time they've implemented a method with an identical signature to the one you wrote:

```
//Utensil.java @version: 2.0
class Utensil{
   public void GetFood(){ //… }
   public void Wash(){
     myDishwasher.add(this);
     //etc…
   }
}
```

The vendor has implemented a **Wash( )** method with complex behavior involving a dishwasher. Given this new capability, people programming with Utensil v2 will have every right to assume that once **Wash( )** has been called, all Utensils will have gone through the dishwasher. But in languages such as Java, the **Wash( )** method in Spork will still be called!

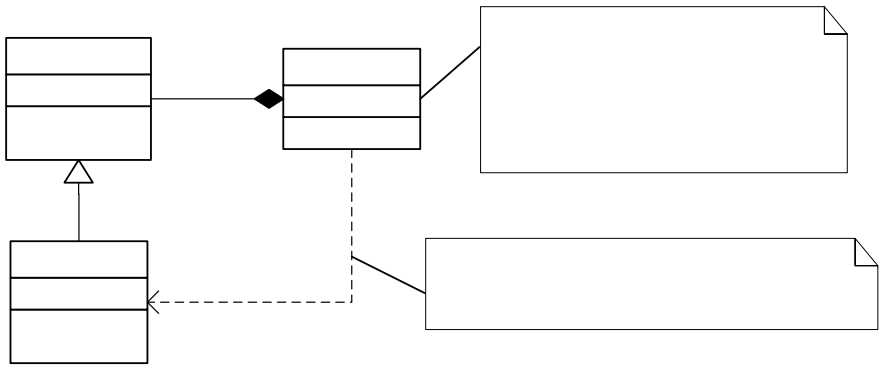


Figure 7-3: Late binding can cause undesired behavior if the base type changes

It may seem highly unlikely that a new version of a base class would "just happen" to have the same name as an end-user's extension, but if you think about it, it's actually kind of surprising it doesn't happen more often, as the number of logical method names for a given category of base class is fairly limited.

In C#, the behavior in **Client**'s **WashAll( )** method would work exactly the way clients expect, with **Utensil**'s dishwasher **Wash( )** being called for all utensils in **myUtensils**, even if one happens to be a **Spork**.

Now let's say you come along and start working on Spork again after upgrading to the version of Utensil that has a **Wash( )** method. When you compile **Spork.cs**, the compiler will say:

```
warning CS0108: The keyword new is required on
'Spork.Wash()' because it hides inherited member
'Utensil.Wash()'
```

At this point, calls to **Utensil.Wash( )** are resolved with the dishwasher washing method, while if you have a handle to a Spork, the napkin-wiping wash method will be called.

```
//:c07:Utensil.cs
using System;

class Utensil {
  public virtual void GetFood(){}
  public void Wash(){
    Console.WriteLine("Washing in a dishwasher");
  }
}

class Fork : Utensil {
  public override void GetFood(){
    Console.WriteLine("Spear");
  }
}

class Spork : Utensil {
  public override void GetFood(){
    Console.WriteLine("Spear OR Scoop!");
  }

  public void Wash(){
```

```
    Console.WriteLine("Wipe with napkin");
  }
}

class Client {
  Utensil[] myUtensils;
  Client(){
    myUtensils = new Utensil[2];
    myUtensils[0] = new Spork();
    myUtensils[1] = new Fork();
  }
  public void WashAll(){
    foreach(Utensil u in myUtensils){
      u.Wash();
    }
  }

  public static void Main(){
    Client c = new Client();
    c.WashAll();
    Spork s = new Spork();
    s.Wash();
  }
}///:~
```

results in the output:

```
Washing in a dishwasher
Washing in a dishwasher
Wipe with napkin
```

In order to remove the warning that **Spork.Wash( )** is hiding the newly minted **Utensil.Wash( )**, we can add the keyword **new** to **Spork**'s declaration:

```
public new void Wash(){ //… etc ...
```

It's even possible for you to have entirely separate method inheritance hierarchies by declaring a method as **new virtual**. Imagine that for version 3 of the Kitchen component, they've created a new type of **Utensil**, **Silverware**, which requires polishing after cleaning. Meanwhile, you've created a new kind of **Spork**, a **SuperSpork**, which also has overridden the base **Spork.Wash( )** method.

The code looks like this:

```
//:c07:Utensil2.cs
using System;

class Utensil {
  public virtual void GetFood(){}
  public virtual void Wash(){
    Console.WriteLine("Washing in a dishwasher");
  }
}

class Silverware : Utensil {
  public override void Wash(){
    base.Wash();
    Console.WriteLine("Polish with silver cleaner");
  }
}

class Fork : Silverware {
  public override void GetFood(){
    Console.WriteLine("Spear");
  }
}

class Spork : Silverware {
  public override void GetFood(){
    Console.WriteLine("Spear OR Scoop!");
  }

  public new virtual void Wash(){
    Console.WriteLine("Wipe with napkin");
  }
}

class SuperSpork : Spork {
  public override void GetFood(){
    Console.WriteLine("Spear AND Scoop");

  }

  public override void Wash(){
    base.Wash();
```

```
    Console.WriteLine("Polish with shirt");
  }
}

class Client {
  Utensil[] myUtensils;
  Client(){
    myUtensils = new Utensil[3];
    myUtensils[0] = new Spork();
    myUtensils[1] = new Fork();
    myUtensils[2] = new SuperSpork();
  }
  public void WashAll(){
    foreach(Utensil u in myUtensils){
      u.Wash();
    }
    Console.WriteLine("All Utensils washed");
  }

  public static void Main(){
    Client c = new Client();
    c.WashAll();
    Spork s = new SuperSpork();
    s.Wash();
  }
}///:~
```

Now, all of our Utensils have been replaced by Silverware and, when **Client.WashAll( )** is called, **Silverware.Wash( )** overloads **Utensil.Wash( )**. (Note that **Silverware.Wash( )** calls **Utensil.Wash( )** using **base.Wash( ),** in the same manner as base constructors can be called.) All **Utensil**s in **Client**'s **myUtensils** array are now washed in a dishwasher and then polished. Note the declaration in **Spork**:

```
public new virtual void Wash(){ //etc }
```

and the declaration in the newly minted **SuperSpork** class:

```
public override void Wash(){ //etc. }
```

When the Client class has a reference to a Utensil such as it does in **WashAll**() (whether the concrete type of that Utensil be a Fork, a Spoon, or a Spork), the **Wash( )** method resolves to the appropriate overloaded method in **Silverware**. When, however, the client has a reference to a Spork or any Spork-subtype, the

Wash( ) method resolves to whatever has overloaded **Spork**'s **new virtual Wash( )**. The output looks like this:

```
Washing in a dishwasher
Polish with silver cleaner
Washing in a dishwasher
Polish with silver cleaner
Washing in a dishwasher
Polish with silver cleaner
All Utensils washed
Wipe with napkin
Polish with shirt
```

And this UML diagram shows the behavior graphically:

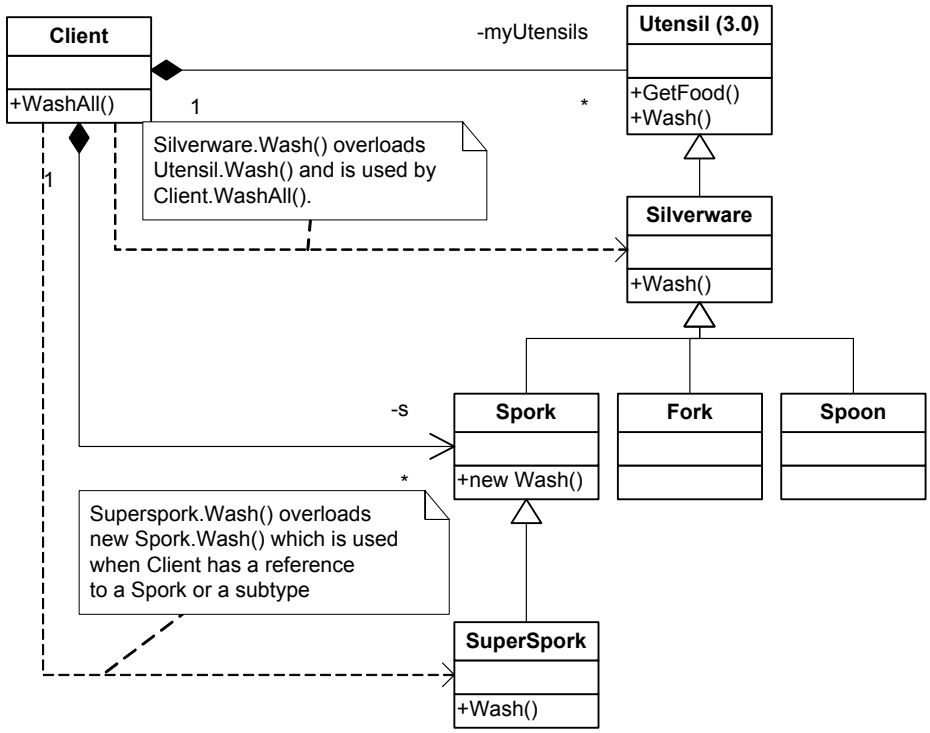


Figure 7-4: C#'s binding model allows fine-tuned control of late-binding

Let's say that you wanted to create a new class **SelfCleansingSuperSpork**, that overloaded both the **Wash( )** method as defined in **Utensil** and the **Wash( )** method as defined in **Spork**. What could you do? You cannot create a single method name that overrides both base methods. As is generally the case,

when faced with a hard programming problem, the answer lies in design, not language syntax. Follow the maxim: boring code, interesting results.

One of the first things that jumps out when considering this problem is that the inheritance hierarchy is getting deep. What we're proposing is that a **SelfCleaningSuperSpork** is-a **SuperSpork** is-a **Spork** is-a **Silverware** is-a **Utensil** is-an **object**. That's six levels of hierarchy – one more than Linnaeus used to classify all living beings in 1735! It's not impossible for a design to have this many layers of inheritance, but in general, one should be dubious of hierarchies of more than two or three levels below **object.**

Bearing in mind that our hierarchy is getting deep, we might also notice that our names are becoming long and unnatural – **SelfCleaningSuperSpork.** While coming up with descriptive names without getting cute is one of the harder tasks in programming – **Execute( ), Run( ),** and **Query( )** are bad, but I've heard a story of a variable labeled **riplvb** because it's initial value happened to be 0x723, or decimal 1827, the year Ludwig van Beethoven died. Something's wrong when a class name becomes a hodge-podge of adjectives. In this case, our names are being used to distinguish between two different properties – the shape of the Utensil (**Fork, Spoon, Spork,** and **SuperSpork**) and the cleaning behavior (**Silverware, Spork,** and **SelfCleaningSuperSpork**).

This is a clue that our design would be better using composition rather than inheritance. As is very often the case, we discover that one of the "vectors of change" is more naturally structural (the shape of the utensil) and that another is more behavioral (the cleaning regimen). We can try out the phrase "A utensil has a cleaning regimen," to see if it sounds right, which indeed it does:

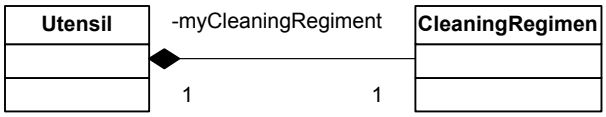


Figure 7-5: Refactoring the design

When a Utensil is constructed, it has a handle to a particular type of cleaning regimen, but its Wash method doesn't have to know the specific subtype of CleaningRegimen it is using:

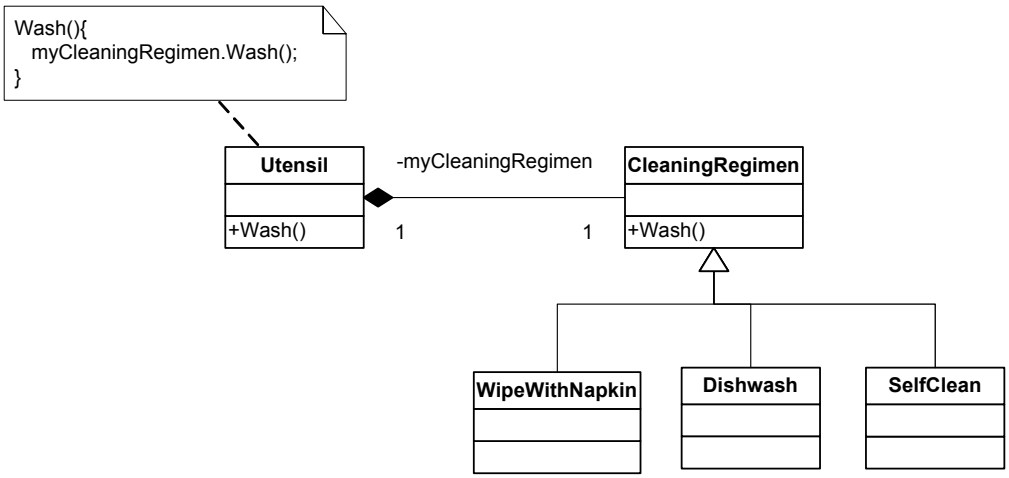


Figure 7-6: The Strategy design pattern

This is called the *Strategy* Pattern and it is, perhaps, the most important of all the design patterns.

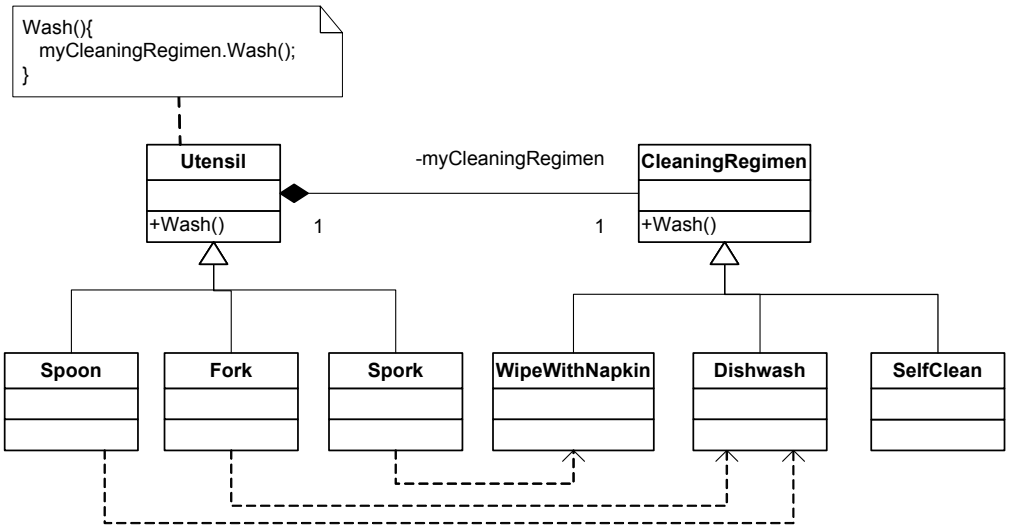


Figure 7-7: Manipulating CleaningRegimens on a per-Utensil basis

This is what the code would look like:

```
//:c07:Utensil3.cs
using System;

class Utensil {
```

```
  CleaningRegimen myCleaningRegimen;
  internal Utensil(CleaningRegimen reg){
    myCleaningRegimen = reg;
  }

  void Wash(){
    myCleaningRegimen.Wash();
  }

  internal virtual void GetFood(){
  }
}

class Fork : Utensil {
  Fork() : base(new Dishwash()){}

  internal override void GetFood(){
    Console.WriteLine("Spear food");
  }
}

class Spoon : Utensil {
  Spoon() : base(new Dishwash()){}
  internal override void GetFood(){
    Console.WriteLine("Scoop food");
  }
}

class Spork : Utensil {
  Spork() : base(new WipeWithNapkin()){}
  internal override void GetFood(){
    Console.WriteLine("Spear or scoop!");
  }
}

abstract class CleaningRegimen {
  internal abstract void Wash();
}

class Dishwash : CleaningRegimen {
  internal override void Wash(){
```

```
    Console.WriteLine("Wash in dishwasher");
  }
}

class WipeWithNapkin : CleaningRegimen {
  internal override void Wash(){
    Console.WriteLine("Wipe with napkin");
  }
}///:~
```

At this point, every type of Utensil has a particular type of CleaningRegimen associated with it, an association which is hard-coded in the constructors of the Utensil subtypes (i.e., **public Spork( ) : base(new WipeWithNapkin( ))**). However, you can see how it would be a trivial matter to totally decouple the **Utensil**'s type of **CleaningRegimen** from the constructor – you could pass in the **CleaningRegimen** from someplace else, choose it randomly, and so forth.

With this design, one can easily achieve our goal of a super utensil that combines multiple cleaning strategies:

```
class SuperSpork : Spork{
    CleaningRegimen secondRegimen;
    public SuperSpork: super(new Dishwash()){
      secondRegimen = new NapkinWash();
    }
    public override void Wash(){
      base.Wash();
      secondRegimen.Wash();
    }
}
```

In this situation, the SuperSpork now has two CleaningRegimens, the normal **myCleaningRegimen** and a new **secondRegimen**. This is the type of flexibility that you can hope to achieve by favoring aggregation over inheritance.

Our original challenge, though, involved a 3rd party Kitchen component that provided the basic design. Without access to the source code, there is no way to implement our improved design. This is one of the things that makes it hard to write components for reuse – "fully baked" components that are easy to use out of the box are often hard to customize and extend, while "construction kit" components that need to be assembled typically can sometimes take a long time to learn.

The const and readonly keywords

We know a CTO who, when reviewing code samples of potential programmers, scans for numeric constants in the code – one strike and the resume goes in the trash. We're happy we never showed him any code for calendar math, because we don't think NUMBER_OF_DAYS_IN_WEEK is clearer than 7. Nevertheless, application code often has lots of data that never changes and C# provides two choices as to how to embody them in code.

The **const** keyword can be applied to value types: sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, bool, char, string, structs and enums. **const** fields are evaluated at compile-time, allowing for marginal performance improvements. For instance:

```
//Number of milliseconds in a day
const long MS_PER_DAY = 1000 * 60 * 60 * 24;
```

will be replaced at compile time with the single value 86,400,000 rather than triggering three multiplications every time it is used.

The **readonly** keyword is more general. It can be applied to any type and is evaluated once — and only once — at runtime. Typically, **readonly** fields are initialized at either the time of class loading (in the case of static fields), or at the time of instance initialization for instance variables. It's not necessary to limit **readonly** fields to values that are essentially constant; you may use a **readonly** field for any data that, once assigned, should be invariant – a person's name or social security number, a network address or port of a host, etc.

readonly does not make an object immutable. When applied to a non-value-type object, **readonly** locks only your reference to the object, not the state of the object itself. Such an object can go through whatever state transitions are programmed into it – properties can be set, it can change its internal state based on calculations, etc. The only thing you can't do is change the reference to the object. This can be seen in this example, which demonstrates **readonly**.

```
//:c07:Composition.cs
using System;
using System.Threading;

public class ReadOnly {
  static readonly DateTime
  timeOfClassLoad = DateTime.Now;
  readonly DateTime
```

```
  timeOfInstanceCreation = DateTime.Now;
  public ReadOnly() {
    Console.WriteLine(
      "Class loaded at {0}, Instance created at {1}",
      timeOfClassLoad, timeOfInstanceCreation);
  }

  //used in second part of program
  static readonly ReadOnly ro = new ReadOnly();
  public int id;
  public int Id{
    get{ return id;}
    set{ id = value;}
  }

  public static void Main(){
    for (int i = 0; i < 10; i++) {
      new ReadOnly();
      Thread.Sleep(1000);
    }
    //Can change member
    ro.Id = 5;
    Console.WriteLine(ro.Id);
    //! Compiler says "a static readonly field
    //cannot be assigned to"
    //ro = new ReadOnly();
  }
}///:~
```

In order to demonstrate how objects created at different times will have different fields, the program uses the Thread.Sleep() method from the Threading namespace, which will be discussed at length in Chapter 16. The class **ReadOnly** contains two **readonly** fields – the static **TimeOfClassLoad** field and the instance variable **timeOfInstanceCreation**. These fields are of type **DateTime**, which is the basic .NET object for counting time. Both fields are initialized from the static DateTime property Now, which represents the system clock.

When the **Main** creates the first **ReadOnly** object and the static fields are initialized as discussed previously, **TimeOfClassLoad** is set once and for all. Then, the instance variable field **timeOfInstanceCreation** is initialized. Finally, the constructor is called, and it prints the value of these two fields to the

console. **Thread.Sleep(1000)** is then used to pause the program for a second (1,000 milliseconds) before creating another **ReadOnly**. The behavior of the program until this point would be no different if these fields were not declared as **readonly**, since we have made no attempt to modify the fields.

That changes in the lines below the loop. In addition to the **readonly DateTime** fields, we have a **static readonly ReadOnly** field labeled **ro** (the class **ReadOnly** contains a reference to an instance of **ReadOnly** –the Singleton design pattern again). We also have a property called **Id**, but note that it is not **readonly**.

(As a review of the discussion in Chapter 5, you should be able to figure out how the values of **ro**'s **timeOfClassLoad** and **timeOfInstanceCreation** will relate to the first **ReadOnly** created in the **Main** loop.)

Although the reference to **ro** is read only, the line ro.Id = 5; demonstrates how it is possible to change the state of a **readonly** reference. What we can't do, though, is shown in the commented-out lines in the example – if we attempt to assign to **ro**, we'll get a compile time error.

The advantage of **readonly** over **const** is that **const**'s compile-time math is immutable. If a class PhysicalConstants had a **public const** that set the speed of light to 300,000 kilometers per second and another class used that for compile-time math:

```
const long KILOMETERS_IN_A_LIGHT_YEAR = PhysicalConstants.C
* 3600 * 24 * DAYS_PER_YEAR
```

the value of KILOMETERS_IN_A_LIGHT_YEAR will be based on the 300,000 value, even if the base class is updated to a more accurate value such as 299,792. This will be true until the class that defined **KILOMETERS_IN_A_LIGHT_YEAR** is recompiled with access to the updated PhysicalConstants class. If the fields were **readonly** though, the value for **KILOMETERS_IN_A_LIGHT_YEAR** would be calculated at runtime, and would not need to be recompiled to properly reflect the latest value of C. Again, this is one of those features which may not seem like a big deal to many application developers, but whose necessity is clear to Microsoft after a decade of "DLL Hell."

Sealed classes

The **readonly** and **const** keywords are used for locking down values and references that should not be changed. Because one has to declare a method as **virtual** in order to be overridden, it is easy to create methods that will not be

modified at runtime. Naturally, there is a way to specify that an entire class be unmodifiable. When a class is declared as **sealed**, no one can derive from it.

There are two main reasons to make a class **sealed**. A **sealed** class is more secure from intentional or unintentional tampering. Additionally, virtual methods executed on a **sealed** class can be replaced with direct function calls, providing a slight performance increase.

```
//:c07:Jurassic.cs
// Sealing a class

class SmallBrain {
}

sealed class Dinosaur {
  internal int i = 7;
  internal int j = 1;
  SmallBrain x = new SmallBrain();
  internal void F() {}
}

//! class Further : Dinosaur {}
// error: Cannot extend sealed class 'Dinosaur'

public class Jurassic {
  public static void Main() {
    Dinosaur n = new Dinosaur();
    n.F();
    n.i = 40;
    n.j++;
  }
}///:~
```

Defining the class as **sealed** simply prevents inheritance—nothing more. However, because it prevents inheritance, all methods in a **sealed** class are implicitly non-**virtual**, since there's no way to override them.

Emphasize virtual functions

It can seem sensible to make as few methods as possible **virtual** and even to declare a class as **sealed**. You might feel that efficiency is very important when using your class and that no one could possibly want to override your methods anyway. Sometimes this is true.

But be careful with your assumptions. In general, it's difficult to anticipate how a class can be reused, especially a general-purpose class. Unless you declare a method as **virtual**, you prevent the possibility of reusing your class through inheritance in some other programmer's project simply because you couldn't imagine it being used that way.

Initialization and class loading

In more traditional languages, programs are loaded all at once as part of the startup process. This is followed by initialization, and then the program begins. The process of initialization in these languages must be carefully controlled so that the order of initialization of **static**s doesn't cause trouble. C++, for example, has problems if one **static** expects another **static** to be valid before the second one has been initialized.

C# doesn't have this problem because it takes a different approach to loading. Because everything in C# is an object, many activities become easier, and this is one of them. As you will learn more fully in the next chapter, the compiled code for a set of related classes exists in their own separate file, called an assembly. That file isn't loaded until the code is needed. In general, you can say that "Class code is loaded at the point of first use." This is often not until the first object of that class is constructed, but loading also occurs when a **static** field or **static** method is accessed.

The point of first use is also where the **static** initialization takes place. All the **static** objects and the **static** code block will be initialized in textual order (that is, the order that you write them down in the class definition) at the point of loading. The **static**s, of course, are initialized only once.

Initialization with inheritance

It's helpful to look at the whole initialization process, including inheritance, to get a full picture of what happens. Consider the following code:

```
//:c07:Beetle.cs
// The full process of initialization.
using System;

class Insect {
  int i = 9;
  internal int j;
```

```
  internal Insect() {
    Prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    Prt("static Insect.x1 initialized");
  internal static int Prt(string s) {
    Console.WriteLine(s);
    return 47;
  }
}

class Beetle : Insect {
  int k = Prt("Beetle.k initialized");
  Beetle() {
    Prt("k = " + k);
    Prt("j = " + j);
  }
  static int x2 =
    Prt("static Beetle.x2 initialized");

  public static void Main() {
    Prt("Beetle constructor");
    Beetle b = new Beetle();
  }
} ///:~
```

The output for this program is:

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
Beetle.k initialized
i = 9, j = 0
k = 47
j = 39
```

The first thing that happens when you run **Beetle** is that you try to access **Beetle.Main( )** (a **static** method), so the loader goes out and finds the compiled code for the **Beetle** class (this happens to be in an assembly called **Beetle.exe**). In the process of loading it, the loader notices that it has a base class (that's what the colon after **class Beetle** says), which it then loads. This will happen whether

or not you're going to make an object of that base class. (Try commenting out the object creation to prove it to yourself.)

If the base class has a base class, that second base class would then be loaded, and so on. Next, the **static** initialization in the root base class (in this case, **Insect**) is performed, and then the next derived class, and so on. This is important because the derived-class static initialization might depend on the base class member being initialized properly.

At this point, the necessary classes have all been loaded so the object can be created. First, all the primitives in this object are set to their default values and the object references are set to **null**—this happens in one fell swoop by setting the memory in the object to binary zero. Then, the base-class fields are initialized in textual order, followed by the fields of the object. After the fields are initialized, the base-class constructor will be called. In this case the call is automatic, but you can also specify the base-class constructor call (by placing a color after the **Beetle( )** constructor and then saying **base( )**). The base class construction goes through the same process in the same order as the derived-class constructor. Finally, the rest of the body of the constructor is executed.

Summary

Both inheritance and composition allow you to create a new type from existing types. Typically, however, you use composition to reuse existing types as part of the underlying implementation of the new type, and inheritance when you want to reuse the interface. Since the derived class has the base-class interface, it can be *upcast* to the base, which is critical for polymorphism, as you'll see in the next chapter.

Despite the strong emphasis on inheritance in object-oriented programming, when you start a design you should generally prefer composition during the first cut and use inheritance only when it is clearly necessary. Composition tends to be more flexible. In addition, by using the added artifice of inheritance with your member type, you can change the exact type, and thus the behavior, of those member objects at run-time. Therefore, you can change the behavior of the composed object at run-time.

Although code reuse through composition and inheritance is helpful for rapid project development, you'll generally want to redesign your class hierarchy before allowing other programmers to become dependent on it. Your goal is a hierarchy in which each class has a specific use and is neither too big (encompassing so much functionality that it's unwieldy to reuse) nor annoyingly small (you can't use it by itself or without adding functionality).

Exercises

1. Create two classes, A and B, with default constructors (empty argument lists) that announce themselves. Inherit a new class called C from A, and create a member of class B inside C. Do not create a constructor for C. Create an object of class C and observe the results.
2. Modify Exercise 1 so that A and B have constructors with arguments instead of default constructors. Write a constructor for C and perform all initialization within C's constructor.
3. Create a simple class. Inside a second class, define a field for an object of the first class. Use lazy initialization to instantiate this object.
4. Inherit a new class from class **Detergent**. Override **Scrub( )** and add a new method called **Sterilize( )**.
5. Take the file **Cartoon.cs** and comment out the constructor for the **Cartoon** class. Explain what happens.
6. Take the file **Chess.cs** and comment out the constructor for the **Chess** class. Explain what happens.
7. Prove that default constructors are created for you by the compiler.
8. Prove that the base-class constructors are (a) always called, and (b) called before derived-class constructors.
9. Create a base class with only a nondefault constructor, and a derived class with both a default and nondefault constructor. In the derived-class constructors, call the base-class constructor.
10. Create a class called **Root** that contains an instance of each of classes (that you also create) named **Component1**, **Component2**, and **Component3**. Derive a class **Stem** from **Root** that also contains an instance of each "component." All classes should have default constructors that print a message about that class.
11. Modify Exercise 10 so that each class only has nondefault constructors.
12. Add a proper hierarchy of **Dispose**( ) methods to all the classes in Exercise 11.

13. Create a class with a method that is overloaded three times. Inherit a new class, add a new overloading of the method, and show that all four methods are available in the derived class.

14. In **Car.cs** add a **Service( )** method to **Engine** and call this method in **Main( )**.

15. Create a class inside a namespace. Your class should contain a **protected** method and a **protected internal** method. Compile this class into a library assembly. Write a new class that tries to call these methods; compile this class into an executable assembly (you'll need to reference the library assembly while compiling, of course). Explain the results. Now inherit from your first class and call the **protected** and **protected internal** methods from this derived class. Compile this derived class into its own assembly and explain the resulting behavior.

16. Create a class called **Amphibian**. From this, inherit a class called **Frog**. Put appropriate methods in the base class. In **Main( )**, create a **Frog** and upcast it to **Amphibian**, and demonstrate that all the methods still work.

17. Modify Exercise 16 so that Frog overrides the method definitions from the base class (provides new definitions using the same method signatures). Note what happens in **Main( )**.

18. Create a class with a method that is not defined as virtual. Inherit from that class and attempt to override that method.

19. Create a **sealed** class and attempt to inherit from it.

20. Prove that class loading takes place only once. Prove that loading may be caused by either the creation of the first instance of that class, or the access of a static member.

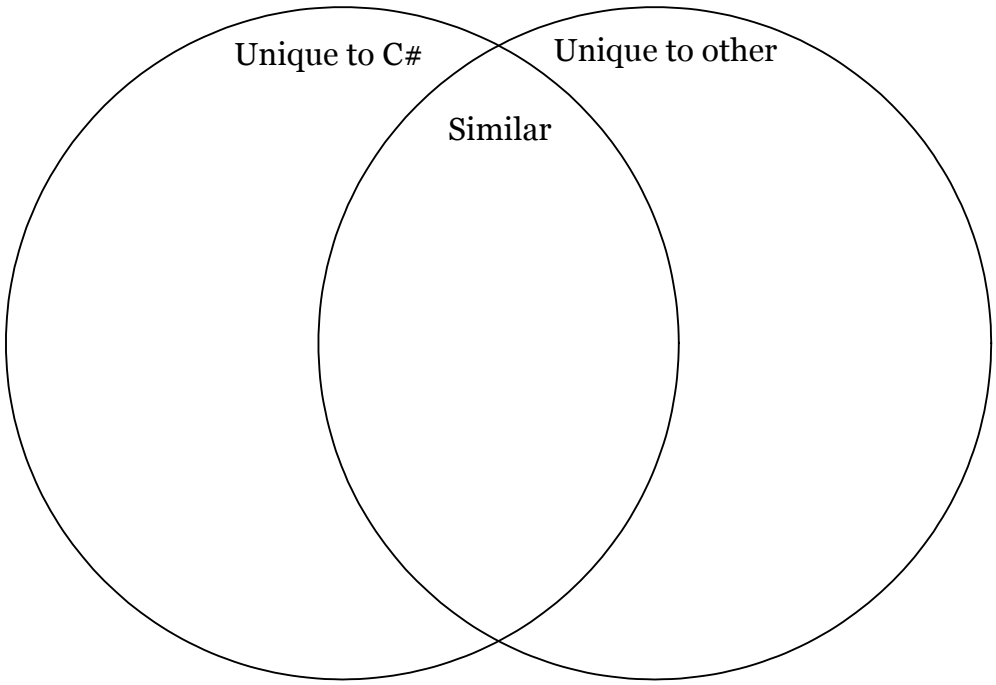
21. In **Beetle.cs**, inherit a specific type of beetle from class **Beetle**, following the same format as the existing classes. Trace and explain the output.

22. Find a way where inheritance can be used fruitfully in the party domain. Implement at least one program that solves a problem by upcasting.

23. Draw a UML class diagram of the party domain, showing inheritance and composition. Place classes that interact often near each other and classes in different namespaces far apart or even on separate pieces of paper.

Consider the task of ensuring that all guests are given a ride home by someone sober or given a place to sleep over. Add classes, namespaces, methods, and data as appropriate.

24. Consider how you would approach the tasks that you have solved in the party domain in the programming language other than C#, with which you are most familiar. Fill in this Venn diagram comparing aspects of the C# approach with how you would do it otherwise:



- Are there aspects unique to one approach that you see as having a major productivity impact?
- What are some important aspects that both approaches share?

8: Interfaces and Implementation

Polymorphism is the next essential feature of an object-oriented programming language after data abstraction. It allows programs to be developed in the form of interacting agreements or "contracts" that specify the behavior, but not the implementation, of classes.

Polymorphism provides a dimension of separation of interface from implementation, to decouple *what* from *how*. Polymorphism allows improved code organization and readability as well as the creation of *extensible* programs that can be "grown" not only during the original creation of the project but also when new features are desired.

Encapsulation creates new data types by combining characteristics and behaviors. Implementation hiding separates the interface from the implementation by making the details **private**. This sort of mechanical organization makes ready sense to someone with a procedural programming background. But polymorphism deals with decoupling in terms of *types*. In the last chapter, you saw how inheritance allows the treatment of an object as its own type *or* its base type. This ability is critical because it allows many types (derived from the same base type) to be treated as if they were one type, and a single piece of code to work on all those different types equally. The polymorphic method call allows one type to express its distinction from another, similar type, as long as they're both derived from the same base type. This distinction is expressed through differences in behavior of the methods that you can call through the base class.

In this chapter, you'll learn about polymorphism (also called *dynamic binding* or *late binding* or *run-time binding*) starting from the basics, with simple examples that strip away everything but the polymorphic behavior of the program.

Upcasting revisited

In Chapter 7 you saw how an object can be used as its own type or as an object of its base type. Taking an object reference and treating it as a reference to its base type is called *upcasting,* because of the way inheritance trees are drawn with the base class at the top.

You also saw a problem arise, which is embodied in the following:

```
//:c08:Music.cs
// Inheritance & upcasting.
using System;

public class Note {
  private int value;
  private Note(int val) { value = val;}
  public static Note
    MIDDLE_C = new Note(0),
    C_SHARP  = new Note(1),
    B_FLAT   = new Note(2);
} // Etc.

public class Instrument {
  public virtual void Play(Note n) {
    Console.WriteLine("Instrument.Play()");
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind : Instrument {
  // Redefine interface method:
  public override void Play(Note n) {
    Console.WriteLine("Wind.Play()");
  }
}

public class Music {
  public static void Tune(Instrument i) {
    // ...
    i.Play(Note.MIDDLE_C);
  }
```

```
  public static void Main() {
    Wind flute = new Wind();
    Tune(flute); // Upcasting
  }
} ///:~
```

The method **Music.Tune( )** accepts an **Instrument** reference, but also anything derived from **Instrument**. In **Main( )**, you can see this happening as a **Wind** reference is passed to **Tune( )**, with no cast necessary. This is acceptable; the interface in **Instrument** must exist in **Wind**, because **Wind** is inherited from **Instrument**. Upcasting from **Wind** to **Instrument** may "narrow" that interface, but it cannot make it anything less than the full interface to **Instrument**.

Forgetting the object type

This program might seem strange to you. Why should anyone intentionally *forget* the type of an object? This is what happens when you upcast, and it seems like it could be much more straightforward if **Tune( )** simply takes a **Wind** reference as its argument. This brings up an essential point: If you did that, you'd need to write a new **Tune( )** for every type of **Instrument** in your system. Suppose we follow this reasoning and add **Stringed** and **Brass** instruments:

```
//:c08:Music2.cs
// Overloading instead of upcasting.
using System;

class Note {
  private int value;
  private Note(int val) { value = val;}
  public static readonly Note
    MIDDLE_C = new Note(0),
    C_SHARP = new Note(1),
    B_FLAT = new Note(2);
} // Etc.

class Instrument {
  internal virtual void Play(Note n) {
    Console.WriteLine("Instrument.Play()");
  }
}

class Wind : Instrument {
```

```
  internal override void Play(Note n) {
    Console.WriteLine("Wind.Play()");
  }
}

class Stringed : Instrument {
  internal override void Play(Note n) {
    Console.WriteLine("Stringed.Play()");
  }
}

class Brass : Instrument {
  internal override void Play(Note n) {
    Console.WriteLine("Brass.Play()");
  }
}

public class Music2 {
  internal static void Tune(Wind i) {
    i.Play(Note.MIDDLE_C);
  }
  internal static void Tune(Stringed i) {
    i.Play(Note.MIDDLE_C);
  }
  internal static void Tune(Brass i) {
    i.Play(Note.MIDDLE_C);
  }
  public static void Main() {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    Tune(flute); // No upcasting
    Tune(violin);
    Tune(frenchHorn);
  }
} ///:~
```

This works, but there's a major drawback: You must write type-specific methods for each new **Instrument** class you add. This means more programming in the first place, but it also means that if you want to add a new method like **Tune( )** or a new type of **Instrument**, you've got a lot of work to do. Add the fact that the

compiler won't give you any error messages if you forget to overload one of your methods and the whole process of working with types becomes unmanageable.

Wouldn't it be much nicer if you could just write a single method that takes the base class as its argument, and not any of the specific derived classes? That is, wouldn't it be nice if you could forget that there are derived classes, and write your code to talk only to the base class?

That's exactly what polymorphism allows you to do. However, most programmers who come from a procedural programming background have a bit of trouble with the way polymorphism works.

The twist

The difficulty with **Music.cs** can be seen by running the program. The output is **Wind.Play( )**. This is clearly the desired output, but it doesn't seem to make sense that it would work that way. Look at the **Tune( )** method:

```
public static void tune(Instrument i) {
  // ...
  i.Play(Note.MIDDLE_C);
}
```

It receives an **Instrument** reference. So how can the compiler possibly know that this **Instrument** reference points to a **Wind** in this case and not a **Brass** or **Stringed**? The compiler can't. To get a deeper understanding of the issue, it's helpful to examine the subject of *binding*.

Method-call binding

Connecting a method call to a method body is called *binding*. When binding is performed before the program is run (by the compiler and linker, if there is one), it's called *early binding*. You might not have heard the term before because it has never been an option with procedural languages. C compilers have only one kind of method call, and that's early binding.

The confusing part of the above program revolves around early binding because the compiler cannot know the correct method to call when it has only an **Instrument** reference.

The solution is called *late binding*, which means that the binding occurs at run-time based on the type of object. Late binding is also called *dynamic binding* or *run-time binding*. When a language implements late binding, there must be some mechanism to determine the type of the object at run-time and to call the

appropriate method. That is, the compiler still doesn't know the object type, but the method-call mechanism finds out and calls the correct method body. The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects.

Obviously, since there's additional behavior at runtime, late binding is a little more time-consuming than early binding. More importantly, if a method is early bound and some other conditions are met, an optimizing compiler may decide not to make a call at all, but instead to place a copy of the method's source code directly into the source code where the call occurs. Such *inlining* may cause the resulting binary code to be a little larger, but can result in significant performance increases in tight loops, especially when the called method is small. Additionally, the contents of an early-bound method can be analyzed and additional optimizations that can never be safely applied to late-bound methods (such as aggressive *code motion* optimizations) may be possible. To give you an idea, a 2001 study[1] showed Fortran-90 running several times as fast as, and sometimes more than an order of magnitude faster than, Java on a series of math-oriented benchmarks (the authors' prototype performance-oriented Java compiler and libraries gave dramatic speedups).Larry ported some of the benchmarks to C# and was disappointed to see results that were very comparable to Java performance[2].

All methods declared as **virtual** or **override** in C# use late binding, otherwise, they use early binding (confirm). This is an irritation but not a big burden. There are two scenarios: either you know that you're going to override a method later on, in which case it's no big deal to add the keyword, or you discover down the road that you need to override a method that you hadn't planned on overriding, which is a significant enough design change to justify a re-examination and recompilation of the base class' code! The one thing you can't do is change the binding from early-bound to late-bound in a component for which you can't perform a recompile because you don't have the source code.

Producing the right behavior

Once you know that virtual method binding in C# happens polymorphically via late binding, you can write your code to talk to the base class and know that all the derived-class cases will work correctly using the same code. Or to put it

---

[1] *The Ninja Project*, Moreira et al., Communications of the ACM 44(10), Oct 2001.

[2] For details, see http://www.ThinkingIn.Net

another way, you "send a message to an object and let the object figure out the right thing to do."

The classic example in OOP is the "shape" example. This is commonly used because it is easy to visualize, but unfortunately it can confuse novice programmers into thinking that OOP is just for graphics programming, which is of course not the case.

The shape example has a base class called **Shape** and various derived types: **Circle**, **Square**, **Triangle**, etc. The reason the example works so well is that it's easy to say "a circle is a type of shape" and be understood. The inheritance diagram shows the relationships:

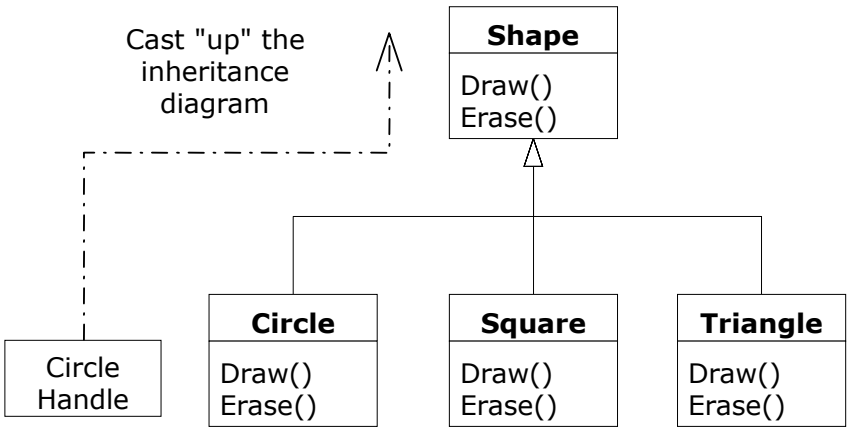


Figure 8-1: Upcasting to Shape

The upcast could occur in a statement as simple as:

```
Shape s = new Circle();
```

Here, a **Circle** object is created and the resulting reference is immediately assigned to a **Shape**, which would seem to be an error (assigning one type to another); and yet it's fine because a **Circle** *is* a **Shape** by inheritance. So the compiler agrees with the statement and doesn't issue an error message.

Suppose you call one of the base-class methods (that have been overridden in the derived classes):

```
s.Draw();
```

Again, you might expect that **Shape**'s **Draw( )** is called because this is, after all, a **Shape** reference—so how could the compiler know to do anything else? And yet the proper **Circle.Draw( )** is called because of late binding (polymorphism).

The following example puts it a slightly different way:

```
//:c08:Shapes.cs
// Polymorphism in C#
using System;

public class Shape {
  internal virtual void Draw() {}
  internal virtual void Erase() {}
}

class Circle : Shape {
  internal override void Draw() {
    Console.WriteLine("Circle.Draw()");
  }
  internal override void Erase() {
    Console.WriteLine("Circle.Erase()");
  }
}

class Square : Shape {
  internal override void Draw() {
    Console.WriteLine("Square.Draw()");
  }
  internal override void Erase() {
    Console.WriteLine("Square.Erase()");
  }
}

class Triangle : Shape {
  internal override void Draw() {
    Console.WriteLine("Triangle.Draw()");
  }
  internal override void Erase() {
    Console.WriteLine("Triangle.Erase()");
  }
}
```

```
public class Shapes {
  static Random rand = new Random();

  public static Shape RandShape() {
    switch (rand.Next(3)) {
    case 0: return new Circle();
    case 1: return new Square();
    case 2: return new Triangle();
    default: return null;
    }
  }
  public static void Main() {
    Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for (int i = 0; i < s.Length;i++)
      s[i] = RandShape();
    // Make polymorphic method calls:
    foreach(Shape aShape in s)
    aShape.Draw();
  }
} ///:~
```

The base class **Shape** establishes the common interface to anything inherited from **Shape**—that is, all shapes can be drawn and erased. The derived classes override these definitions to provide unique behavior for each specific type of shape.

The main class **Shapes** contains a **static** method **RandShape( )** that produces a reference to a randomly-selected **Shape** object each time you call it. Note that the upcasting happens in the **return** statements, each of which takes a reference to a **Circle**, **Square**, or **Triangle** and sends it out of the method as the return type, **Shape**. So whenever you call this method you never get a chance to see what specific type it is, since you always get back a plain **Shape** reference.

Main( ) contains an array of **Shape** references filled through calls to **RandShape( )**. At this point you know you have **Shape**s, but you don't know anything more specific than that (and neither does the compiler). However, when you step through this array and call **Draw( )** for each one, the correct type-specific behavior magically occurs, as you can see from one output example:

```
Circle.Draw()
Triangle.Draw()
Circle.Draw()
```

```
Circle.Draw()
Circle.Draw()
Square.Draw()
Triangle.Draw()
Square.Draw()
Square.Draw()
```

Of course, since the shapes are all chosen randomly each time, your runs will have different results. The point of choosing the shapes randomly is to drive home the understanding that the compiler can have no special knowledge that allows it to make the correct calls at compile-time. All the calls to **Draw( )** are made through dynamic binding.

Extensibility

Now let's return to the musical instrument example. Because of polymorphism, you can add as many new types as you want to the system without changing the **Tune( )** method. In a well-designed OOP program, most or all of your methods will follow the model of **Tune( )** and communicate only with the base-class interface. Such a program is *extensible* because you can add new functionality by inheriting new data types from the common base class. The methods that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

Consider what happens if you take the instrument example and add more methods in the base class and a number of new classes. Here's the diagram:

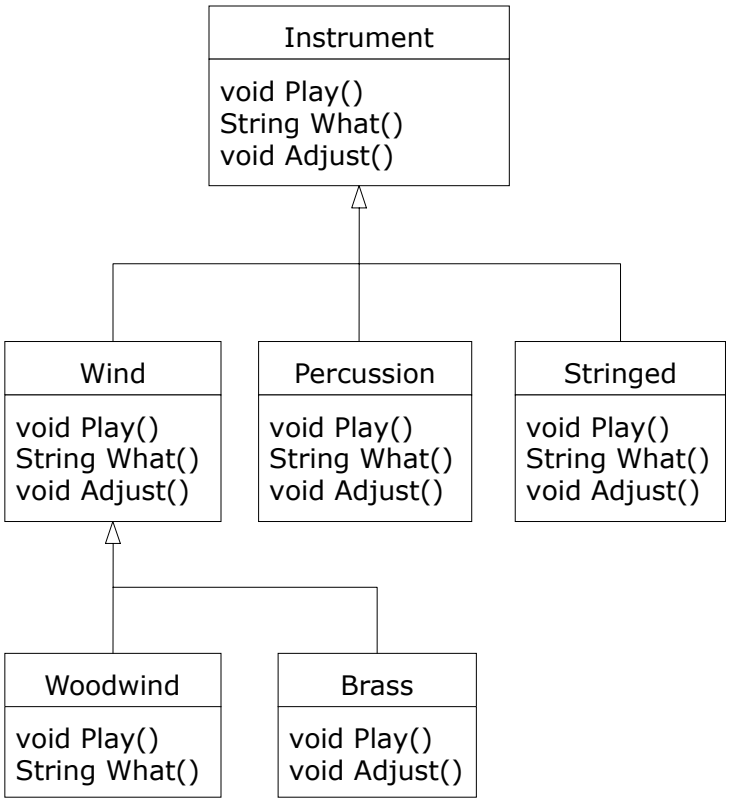


Figure 8-2: Despite increased complexity, old code works

All these new classes work correctly with the old, unchanged **Tune( )** method. Even if **Tune( )** is in a separate file and new methods are added to the interface of **Instrument**, **Tune( )** works correctly without recompilation. Here is the implementation of the above diagram:

```
//:c08:Music3.cs
// An extensible program.
using System;

class Instrument {
  public virtual void Play() {
    Console.WriteLine("Instrument.Play()");
  }
  public virtual string What() {
```

```
    return "Instrument";
  }
  public virtual void Adjust() {}
}

class Wind : Instrument {
  public override void Play() {
    Console.WriteLine("Wind.Play()");
  }
  public override string What() { return "Wind";}
  public override void Adjust() {}
}

class Percussion : Instrument {
  public override void Play() {
    Console.WriteLine("Percussion.Play()");
  }
  public override string What() {
    return "Percussion";}
  public override void Adjust() {}
}

class Stringed : Instrument {
  public override void Play() {
    Console.WriteLine("stringed.Play()");
  }
  public override string What() { return "Sstringed";}
  public override void Adjust() {}
}

class Brass : Wind {
  public override void Play() {
    Console.WriteLine("Brass.Play()");
  }
  public override void Adjust() {
    Console.WriteLine("Brass.Adjust()");
  }
}

class Woodwind : Wind {
  public override void Play() {
```

```
    Console.WriteLine("Woodwind.Play()");
  }
  public override string What() { return "Woodwind";}
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void Tune(Instrument i) {
    // ...
    i.Play();
  }
  static void TuneAll(Instrument[] e) {
    foreach(Instrument i in e)
    Tune(i);
  }
  public static void Main() {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    TuneAll(orchestra);
  }
} ///:~
```

Technically you don't need those methods (in this or any of the later Music examples), but I think it gets confusing – especially later on when you get into abstract classes and interfaces. They can also be used to make the point that not all virtual methods need to be overridden, but if you leave the examples as they are, at least point it out, because otherwise it leaves the reader wondering why you chose to do that.The new methods are **What( )**, which returns a **String** reference with a description of the class, and **Adjust( )**, which provides some way to adjust each instrument.

In **Main( )**, when you place something inside the **Instrument** array you automatically upcast to **Instrument**.

You can see that the **Tune( )** method is blissfully ignorant of all the code changes that have happened around it, and yet it works correctly. This is exactly what polymorphism is supposed to provide. Your code changes don't cause damage to parts of the program that should not be affected. Put another way, polymorphism is one of the most important techniques that allow the programmer to "separate the things that change from the things that stay the same."

Static methods cannot be virtual

As you know, there is a difference between a class (the type) and an object (an instance of that class). Data and methods can either be associated with the class (**static** data and methods) or with individual objects ("instance" data and methods). Unfortunately, polymorphism does not work with **static** methods. This is not a logical consequence of object orientation, it is a result of how polymorphism is implemented.

Take sound equipment, where there are several types of components (CD players and so forth) that you might own. Each type of component has a number of channels that is characteristic: all **CdPlayer**s have two channels and all Dolby decoders have "5+1" channels. On the other hand, adjusting the sound is something that is done polymorphically to individual components: the ways you can adjust the tone from CD players are different than the ways you can adjust a home theater tuner, but when an adjustment is done, it applies to **this** particular **CdPlayer** or **DolbyDecoder**, not to *every* instance of the class.

According to our discussion of polymorphism, it would seem logical that the way one would declare these two methods in the base class would be:

```
virtual static void SayChannel(){ … }
virtual void AdjustSound(){ … }
```

And then we would override them in subtypes with:

```
override static void SayChannel(){ … }
override void AdjustSound(){ … }
```

But the compiler refuses to compile static methods marked **virtual.** Instead, we have to write code such as this:

```
//:c08:StaticNonPolymorphism.cs
//No polymorphism of static methods
using System;

class SoundEquipment {
```

```
  //! static virtual void GetChannels(){
  internal static void SayChannels(){
    Console.WriteLine("I don't know how many");
  }

  internal virtual void AdjustSound(){
    Console.WriteLine("No default adjustment");
  }

  public static void Main(){
    SoundEquipment[] components =
      { new CdPlayer(), new DolbyDecoder()};
    foreach(SoundEquipment c in components){
      //! Console.WriteLine(c.GetChannels());
      SoundEquipment.SayChannels();
      c.AdjustSound();
    }
  }
}

class CdPlayer: SoundEquipment {
  //!static override void SayChannels(){
  static new void SayChannels(){
    Console.WriteLine(
      "All CD players have 2 channels");
  }

  internal override void AdjustSound(){
    Console.WriteLine("Adjusting total volume");
  }
}

class DolbyDecoder : SoundEquipment {
  //! static override void SayChannels(){
  static new void SayChannels(){
    Console.WriteLine(
      "All DolbyDecoders have 5+1 channels");
  }

  internal override void AdjustSound(){
    Console.WriteLine("Adjusting effects channel");
```

```
  }
}///:~
```

The **SoundEquipment.Main( )** method creates a **CdPlayer** and a **DolbyDecoder** and upcasts the result into a **SoundEquipment[ ]** array. It then calls the **static SoundEquipment.SayChannels( )** method and the virtual **SoundEquipment.AdjustSound( )** method. The **SoundEquipment.AdjustSound( )** virtual method call works as we desire, late-binding to our particular **CdPlayer** and **DolbyDecoder** objects, but the **SoundEquipment.SayChannels( )** does not. The output is:

```
I don't know how many
Adjusting total volume
I don't know how many
Adjusting effects channel
```

The many benefits of overriding method calls are simply not available to **static** methods. The way that virtual method calls are implemented requires a reference to **this** and the hassle of a different implementation is great enough that the lack of **static virtual** methods is allowed to pass.

Overriding vs. overloading

Let's take a different look at the first example in this chapter. In the following program, the interface of the method **Play( )** is changed in the process of overriding it, which means that you haven't *overridden* the method, but instead *overloaded* it. The compiler allows you to overload methods so it gives no complaint. But the behavior is probably not what you want. Here's the example:

```
//:c08:WindError.cs
// Accidentally changing the interface.
using System;

public class NoteX {
  public const int
    MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

public class InstrumentX {
  public void Play(int NoteX) {
    Console.WriteLine("InstrumentX.Play()");
  }
}
```

```
public class WindX : InstrumentX {
  // OOPS! Changes the method interface:
  public void Play(NoteX n) {
    Console.WriteLine("WindX.Play(NoteX n)");
  }
}

public class WindError {
  public static void Tune(InstrumentX i) {
    // ...
    i.Play(NoteX.MIDDLE_C);
  }
  public static void Main() {
    WindX flute = new WindX();
    Tune(flute); // Not the desired behavior!
  }
} ///:~
```

There's another confusing aspect thrown in here. In **InstrumentX**, the **Play( )** method takes an **int** that has the identifier **NoteX**. That is, even though **NoteX** is a class name, it can also be used as an identifier without complaint. But in **WindX**, **Play( )** takes a **NoteX** reference that has an identifier **n.** (Although you could even say **Play(NoteX NoteX)** without an error.) Thus it appears that the programmer intended to override **Play( )** but mistyped the method a bit. The compiler, however, assumed that an overload and not an override was intended. Note that if you follow the standard C# naming convention, the argument identifier would be **noteX** (lowercase 'n'), which would distinguish it from the class name.

In **Tune**, the **InstrumentX i** is sent the **Play( )** message, with one of **NoteX**'s members (**MIDDLE_C**) as an argument. Since **NoteX** contains **int** definitions, this means that the **int** version of the now-overloaded **Play( )** method is called, and since that has *not* been overridden the base-class version is used.

The output is:

```
InstrumentX.Play()
```

This certainly doesn't appear to be a polymorphic method call. Once you understand what's happening, you can fix the problem fairly easily, but imagine how difficult it might be to find the bug if it's buried in a program of significant size.

Operator overloading

In C#, you can override and overload operators (e.g., '+', '/', etc.). Some people do not like operator overloading, arguing that operator overloading is confusing for relatively little benefit. Certainly it's true that you should think twice before overloading an operator; operators carry a lot of baggage in terms of expected behavior and, when used, have a tendency to be overlooked in future code reviews. When thought out, though, operator overloading definitely makes code easier to read and write.

To overload an operator, you declare a **static** method that takes, as its first argument, a reference to your type. For unary operators, which apply to a single operator, this is the only argument that you need and the return type of the method must be the same type. The keyword **operator** alerts the compiler that you're creating an overloaded function. This example overloads the '++' unary operator:

```
//:c08:Life.cs
//Demonstrates unary operator overloading
using System;

enum LifeState {
  Birth, School, Work, Death
};

class Life {
  LifeState state;

  Life(){
    state = LifeState.Birth;
  }
  public static Life operator ++(Life l){
    if (l.state != LifeState.Death) {
      l.state++;
    } else {
      Console.WriteLine("Still dead.");
    }
    return l;
  }

  public static void Main(){
    Life myLife = new Life();
```

```
    for (int i = 0; i < 4; i++) {
      Console.WriteLine(myLife.state);
      //Following call uses operator overloading
      myLife++;
    }
  }
}///:~
```

First, we specify the gamut of possible **LifeState**s[3] and, in the **Life( )** constructor, we set the local **LifeState** to **LifeState.Birth**. The next line:

```
public static Life operator ++(Life l)
```

overloads the ++ operator so that it moves inexorably forward until the **Life** is in **LifeState.Death**.

Although the first argument and the return type must be the same as the class in which the operator is overloaded, if you overload an operator in a class from which others descend, you can return objects of different subtypes:

```
//:c08:Canines.cs
//Demonstrates polymorphic operator overloading
using System;

class Canine {
  public virtual void Speak(){}

  public virtual Canine Grow(){ return this;}

  public static void Main(){
    Canine c = new Puppy();
    c.Speak();
    c++;
    c.Speak();
    c++;
    c.Speak();
  }

  public static Canine operator++(Canine c){
    return c.Grow();
```

[3] At least according to the band The Godfathers.

```
  }
}

class Puppy : Canine {
  public override void Speak(){
    Console.WriteLine("Yip!");
  }

  public override Canine Grow(){
    return new Dog();
  }
}

class Dog : Canine {
  public override void Speak(){
    Console.WriteLine("Whoof!");
  }

  public override Canine Grow(){
    Console.WriteLine("Already fully grown");
    return this;
  }
}///:~
```

The ++ operator is overloaded within the **Canine** class, from which **Puppy** and **Dog** descend. If the argument to the ++ operator is a **Canine** that happens to be a **Puppy**, the call to **c.Grow( )** will be resolved by **Puppy.Grow( )**, which returns a **Dog.**

Figure 8-3 illustrates this program's behavior with a UML *Sequence Diagram*. While class diagrams are helpful for illustrating the static structure of a collection of classes, sequence diagrams are helpful when talking about the dynamic behavior of a set of objects. A sequence diagram is read from the top downward, as time increases. Objects of interest are arranged horizontally, with each object's lifespan denoted by a vertical dashed line. A method call is represented by an arrow pointing to the receiving object and the duration of the method call is represented by a thin box on the object's lifeline. Return values are shown using dashed lines. This diagram uses a non-standard convention by showing the names of virtual method calls in italic.

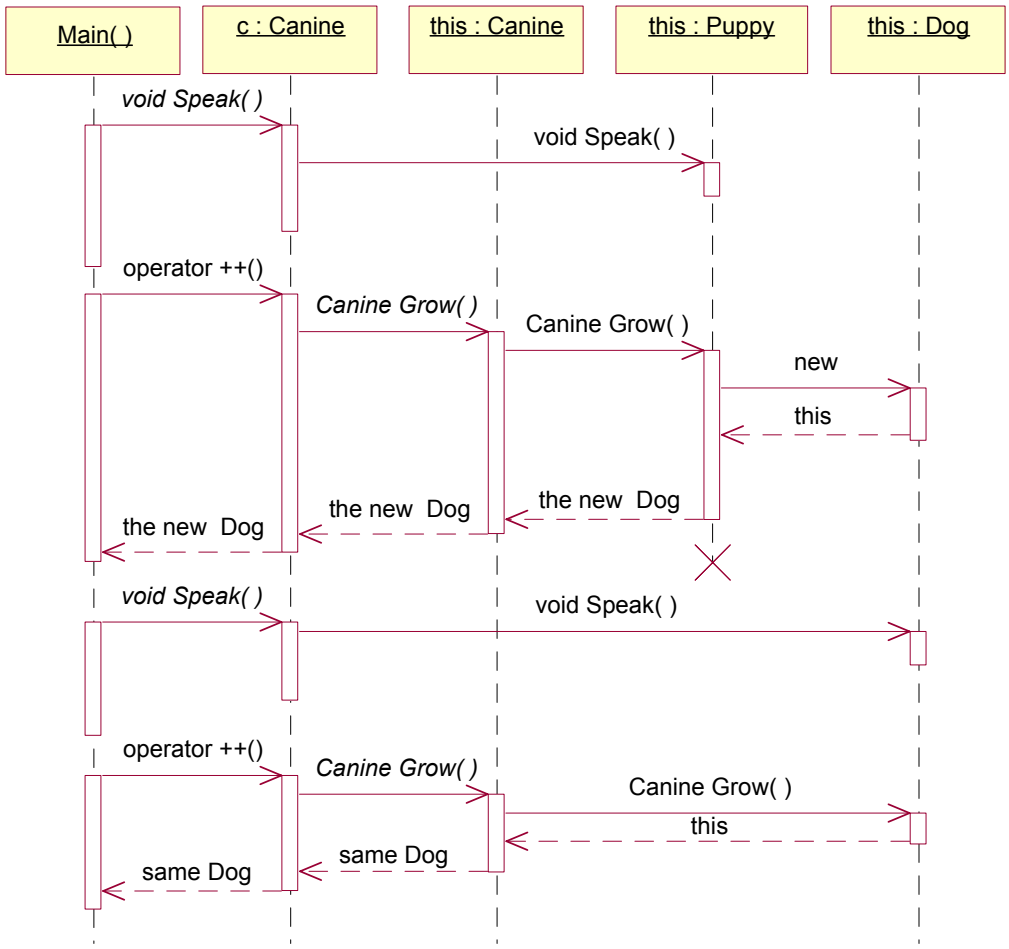


Figure 8-3: Virtual calls to Canine.Grow( ) and Speak( )

The first time **Canine.Main( )** calls the virtual **Canine.Speak( )** method, it is resolved by the **Canine this**, which at this point is a **Puppy**. Similarly, when the ++ operator is called, its first argument is a **Canine** which happens to be a **Puppy**. Thus, the call in **operator++** to **Canine.Grow( )** is resolved by the **Puppy.Grow( )** override. **Puppy.Grow( )** creates a new **Dog** object and returns a reference to it. Since the **Puppy** is no longer referenced, it is now eligible for garbage collection, as indicated on the diagram by the end of the **Puppy**'s object lifeline.

If you look at the **Main( )** method, you'll see that there is no assignment of the results of the ++ operator. Rather, the **Canine** referenced by the **c** variable has

changed from a reference to a **Puppy** to a reference to a **Dog** as part of the application of the ++ operator; you can see how this might be surprising to someone just viewing the **Main( )** method and why the logic of an overloaded operator should be apparent. After the **Puppy** reference has been changed into a **Dog** reference by the first application of the ++ operator, subsequent calls to the virtual method **Canine.Speak( )** will be resolved by **Dog.Speak( )**, as will be virtual calls to **Canine.Grow( )**. The diagram illustrates these behaviors, too.

Multiargument operator overloading

Binary operators are those which take two arguments. One of the two arguments must be the type of the containing class, the other argument can be of any type, allowing full overloading. This example allows either **Fan**s or **Player**s to be added to an **Arena**.

```
//:c08:Arena.cs
//Demonstrates binary operator overloading
using System;

class Arena {
  public static Arena
  operator+(Arena a, Player p){
    a.AddPlayer(p);
    return a;
  }

  public static Arena
  operator+(Arena a, Fan f){
    a.AddFan(f);
    return a;
  }

  void AddPlayer(Player p){
    Console.WriteLine("Player added");
  }

  void AddFan(Fan f){
    Console.WriteLine("Fan added");
  }

  public static void Main(){
    Arena a = new Arena();
```

```
    Fan f = new Fan();
    //Normal form
    a = a + f;

    Player p = new Player();
    //Also works
    a += p;
  }
}

class Player {
}

class Fan {
}///:~
```

The + operator is overloaded twice; both are **static operator** methods that take an **Arena** as the first argument. One overload accepts a **Player** as the second argument, and the other takes a **Fan** object. s are similar; they call an instance method, **Arena.AddFan( )** or **Arena.AddPlayer( )** on their **Arena** argument and return the result.

The static **Arena.Main( )** method creates an **Arena** and a **Fan** and uses the normal form **a = a + f** to add the **Fan**. Then, **Main( )** creates a **Player** and uses **a += p** to add it; in C#, += is not an atomic operator but is simply a combination of the addition and assignment operators (a subtlety that will be revisited in Chapter 16's discussion of threading).

Explicit and implicit type conversions

Among the most common uses of operator overloading is implementing conversions between types. If it is impossible for data to be lost during the conversion, the conversion can be specified as **implicit** and the conversion will not require a cast. If, on the other hand, data *may* be lost, the conversion should be marked as **explicit**, and a client programmer attempting the conversion will need to make a cast.

The operator that one overloads for a conversion is of the form:

```
public static implicitOrExplicit
operator TypeConvertedTo(TypeConvertedFrom){…}
```

where *implicitOrExplicit* is either **implicit** or **explicit**. Although it's easy enough to cast the value of an **enum** to an **int**, we can remove even that burden from users of the **Day** class in this example:

```
//:c08:DayOfWeek.cs
using System;

class Day {
  enum dow {
    Sunday = 0, Monday = 1, Tuesday = 2,
    Wednesday = 3, Thursday = 4, Friday = 5,
    Saturday = 6
  }

  dow dayOfWeek;
  dow DayOfWeek{
    get { return dayOfWeek;}
  }


  Day(int i){
    dayOfWeek = (dow) (i % 7);
  }

  public static explicit operator Day(int i){
    Day d = new Day(i);
    return d;
  }

  //Returns 0 (Sun) - 6 (Fri)
  public static implicit operator int(Day d){
    return(int) d.DayOfWeek;
  }

  public static void Main(){
    //Calls explicit operator Day(int i)
    Day d = (Day) 24;
    Console.WriteLine(d.DayOfWeek);
    //Calls implicit operator int(Day d)
    int iOfWeek = d;
    Console.WriteLine(iOfWeek);
```

```
  }
}///:~
```

The **Day** class overloads the cast from an **int** into a **Day**. Because some data is lost in the conversion (the **int** is converted by modulo arithmetic), the conversion is marked as **explicit**. Conversely, since no data is lost converting from **Day** to **int**, that overload is marked as **implicit**. Both overloads are demonstrated in **Day.Main( )** and, although a cast is needed to convert an **int** into a **Day**, none is needed for the reverse.

Operator overloading design guidelines

If an operator overload's meaning isn't obvious, you shouldn't use operator overloading. Overloading the ++ operator to mean "Increase the object's age" is not obvious and is a bad design choice. As a matter of fact, coming up with "obvious" operator overloads is so difficult that it's the primary argument *against* operator overloading – 90% of the discussions of operator overloading use imaginary numbers as their example because imaginary numbers are one of the few types that clearly pass the "obvious" test.

Operator overloading is not guaranteed to exist in all .NET languages. This means that you must either forego the possibility that your class will be used by a language other than C# (a choice that undermines .NET's fundamental value proposition) or create an equivalent named method that exposes the functionality for other languages.

A general design principle is that classes should have *symmetric interfaces*. This means that methods will often be paired with their logical opposites: if you write an **On( )** method, you should write an **Off( )**, **TurnRight( )** implies a **TurnLeft( ),** etc.. Most operators have an opposite, so if you overload + (the plus operator), you should overload - (the minus operator).

Abstract classes and methods

In all the instrument examples, the methods in the base class **Instrument** were always "dummy" methods. If these methods are ever called, you've done something wrong. That's because the intent of **Instrument** is to create a *common interface* for all the classes derived from it.

The only reason to establish this common interface is so it can be expressed differently for each different subtype. It establishes a basic form, so you can say what's in common with all the derived classes. Another way of saying this is to

call **Instrument** an *abstract base class* (or simply an *abstract class*). You create an abstract class when you want to manipulate a set of classes through this common interface. All derived-class methods that match the signature of the base-class declaration will be called using the dynamic binding mechanism. (However, as seen in the last section, if the method's name is the same as the base class but the arguments are different, you've got overloading, which probably isn't what you want.)

If you have an abstract class like **Instrument**, objects of that class almost always have no meaning. That is, **Instrument** is meant to express only the interface, and not a particular implementation, so creating an **Instrument** object makes no sense, and you'll probably want to prevent the user from doing it. This can be accomplished by making all the methods in **Instrument** print error messages, but that delays the information until run-time and requires reliable exhaustive testing on the user's part. It's always better to catch problems at compile-time.

C# provides a mechanism for doing this called the *abstract method*[4]. This is a method that is incomplete; it has only a declaration and no method body. Here is the syntax for an abstract method declaration:

```
abstract void F();
```

A class containing abstract methods is called an *abstract class*. If a class contains one or more abstract methods, the class must be qualified as **abstract**. (Otherwise, the compiler gives you an error message.)

There's no need to qualify **abstract** methods as **virtual**, as they are always resolved with late binding.

If an abstract class is incomplete, what is the compiler supposed to do when someone tries to instantiate an object of that class? It cannot safely create an object of an abstract class, so you get an error message from the compiler. This way the compiler ensures the purity of the abstract class, and you don't need to worry about misusing it.

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don't (and you may choose not to), then the derived class is also abstract and the compiler will force you to qualify *that* class with the **abstract** keyword.

---

[4] For C++ programmers, this is the analogue of C++'s *pure virtual function*.

It's possible to create a class as **abstract** without including any **abstract** methods. This is useful when you've got a class in which it doesn't make sense to have any **abstract** methods, and yet you want to prevent any instances of that class.

The **Instrument** class can easily be turned into an **abstract** class. Only some of the methods will be **abstract**, since making a class abstract doesn't force you to make all the methods **abstract**. Here's what it looks like:

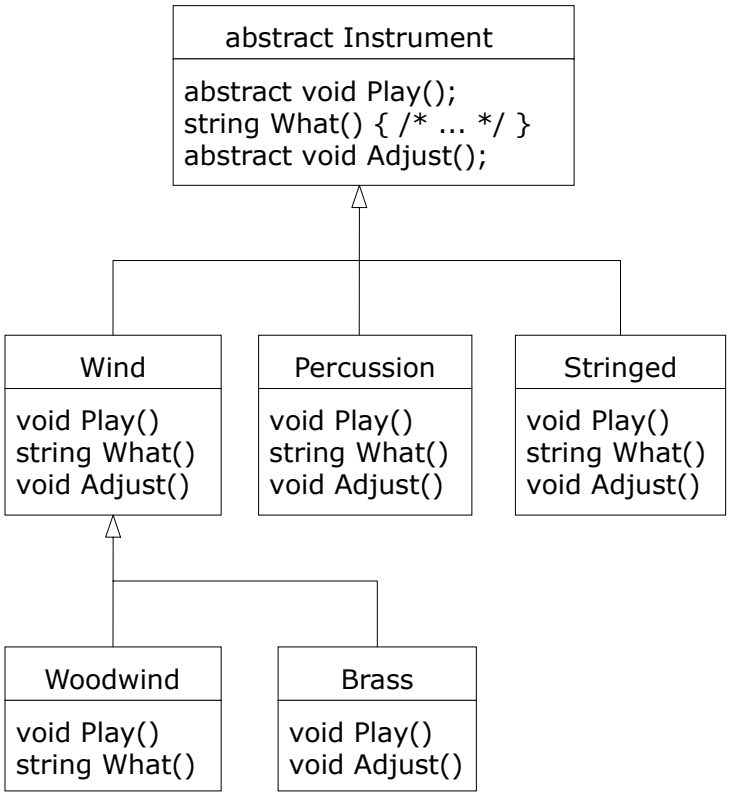


Figure 8-4: Abstract classes provide shared behavior, but cannot be instantiated

Here's the orchestra example modified to use **abstract** classes and methods:

```
//:c08:Music4.cs
// An extensible program.
using System;
```

```
abstract class Instrument {
  public abstract void Play();
  public virtual string What() {
    return "Instrument";
  }
  public abstract void Adjust();
}

class Wind : Instrument {
  public override void Play() {
    Console.WriteLine("Wind.Play()");
  }
  public override string What() { return "Wind";}
  public override void Adjust() {}
}

class Percussion : Instrument {
  public override void Play() {
    Console.WriteLine("Percussion.Play()");
  }
  public override string What() {
    return "Percussion";}
  public override void Adjust() {}
}

class Stringed : Instrument {
  public override void Play() {
    Console.WriteLine("stringed.Play()");
  }
  public override string What() { return "Sstringed";}
  public override void Adjust() {}
}

class Brass : Wind {
  public override void Play() {
    Console.WriteLine("Brass.Play()");
  }
  public override void Adjust() {
    Console.WriteLine("Brass.Adjust()");
  }
}
```

```
class Woodwind : Wind {
  public override void Play() {
    Console.WriteLine("Woodwind.Play()");
  }
  public override string What() { return "Woodwind";}
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void Tune(Instrument i) {
    // ...
    i.Play();
  }
  static void TuneAll(Instrument[] e) {
    foreach(Instrument i in e)
    Tune(i);
  }
  public static void Main() {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    TuneAll(orchestra);
  }
} ///:~
```

You can see that there's really no change except in the base class.

It's helpful to create **abstract** classes and methods because they make the abstractness of a class explicit, and tell both the user and the compiler how it was intended to be used.

Constructors and polymorphism

As usual, constructors are different from other kinds of methods. This is also true when polymorphism is involved. Even though constructors are not polymorphic

(although you can have a kind of "virtual constructor," as you will see in Chapter 13), it's important to understand the way constructors work in complex hierarchies and with polymorphism. This understanding will help you avoid unpleasant entanglements.

Order of constructor calls

The order of constructor calls was briefly discussed in Chapter 7, but that was before polymorphism was introduced.

A constructor for the base class is always called in the constructor for a derived class, chaining up the inheritance hierarchy so that a constructor for every base class is called. This makes sense because the constructor has a special job: to see that the object is built properly. A derived class has access to its own members only, and not to **private** members of the base class. Only the base-class constructor has the proper knowledge and access to initialize its own elements. Therefore, it's essential that all constructors get called, otherwise the entire object wouldn't be consistently constructed. That's why the compiler enforces a constructor call for every portion of a derived class. It will silently call the default constructor if you don't explicitly call a base-class constructor in the derived-class constructor body. If there is no default constructor, the compiler will complain. (In the case where a class has no constructors, the compiler will automatically synthesize a default constructor.)

Let's take a look at an example that shows the effects of composition, inheritance, and polymorphism on the order of construction:

```
//:c08:Sandwich.cs
// Order of constructor calls.
using System;

public class Meal {
  internal Meal() { Console.WriteLine("Meal()");}
}

public class Bread {
  internal Bread() { Console.WriteLine("Bread()");}
}

public class Cheese {
  internal Cheese() { Console.WriteLine("Cheese()");}
}
```

```
public class Lettuce {
  internal Lettuce(){ Console.WriteLine("Lettuce()");}
}

public class Lunch : Meal {
  internal Lunch() { Console.WriteLine("Lunch()");}
}

public class PortableLunch : Lunch {
  internal PortableLunch() {
    Console.WriteLine("PortableLunch()");
  }
}

public class Sandwich : PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  internal Sandwich() {
    Console.WriteLine("Sandwich()");
  }
  public static void Main() {
    new Sandwich();
  }
} ///:~
```

This example creates a complex class out of other classes, and each class has a constructor that announces itself. The important class is **Sandwich**, which reflects three levels of inheritance (four, if you count the implicit inheritance from **object**) and three member objects. When a **Sandwich** object is created in **Main( )**, the output is:

```
Bread()
Cheese()
Lettuce()
Meal()
Lunch()
PortableLunch()
Sandwich()
```

This means that the order of constructor calls for a complex object is as follows:

1. Member initializers are called in the order of declaration.

2. The base-class constructor is called. This step is repeated recursively such that the root of the hierarchy is constructed first, followed by the next-derived class, etc., until the most-derived class is reached.

3. The body of the derived-class constructor is called.

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any **public**, **protected,** or **internal** members of the base class. This means that you must be able to assume that all the members of the base class are valid when you're in the derived class. In a normal method, construction has already taken place, so all the members of all parts of the object have been built. Inside the constructor, however, you must be able to assume that all members that you use have been built. The only way to guarantee this is for the base-class constructor to be called first. Then when you're in the derived-class constructor, all the members you can access in the base class have been initialized. "Knowing that all members are valid" inside the constructor is also the reason that, whenever possible, you should initialize all member objects (that is, objects placed in the class using composition) at their point of definition in the class (e.g., **b**, **c,** and **l** in the example above). If you follow this practice, you will help ensure that all base class members *and* member objects of the current object have been initialized. Unfortunately, this doesn't handle every case, as you will see in the next section.

Behavior of polymorphic methods inside constructors

The hierarchy of constructor calls brings up an interesting dilemma. What happens if you're inside a constructor and you call a dynamically bound method of the object being constructed? Inside an ordinary method you can imagine what will happen—the dynamically bound call is resolved at run-time because the object cannot know whether it belongs to the class that the method is in or some class derived from it. For consistency, you might think this is what should happen inside constructors.

This is not exactly the case. If you call a dynamically bound method inside a constructor, the overridden definition for that method is used. However, the *effect* can be rather unexpected, and can conceal some difficult-to-find bugs.

Conceptually, the constructor's job is to bring the object into existence (which is hardly an ordinary feat). Inside any constructor, the entire object might be only partially formed—you can know only that the base-class objects have been initialized, but you cannot know which classes are inherited from you. A dynamically bound method call, however, reaches "outward" into the inheritance

hierarchy. It calls a method in a derived class. If you do this inside a constructor, you call a method that might manipulate members that haven't been initialized yet—a sure recipe for disaster.

You can see the problem in the following example:

```
//:c08:PolyConstructors.cs
// Constructors and polymorphism
// don't produce what you might expect.
using System;

abstract class Glyph {
  protected abstract void Draw();
  internal Glyph() {
    Console.WriteLine("Glyph() before draw()");
    Draw();
    Console.WriteLine("Glyph() after draw()");
  }
}

class RoundGlyph : Glyph {
  int radius = 1;
  int thickness;
  internal RoundGlyph(int r) {
    radius = r;
    thickness = 2;
    Console.WriteLine("RoundGlyph.RoundGlyph(), "
      + "radius = {0} thickness  ={1}",
      + radius, thickness);
  }
  protected override void Draw() {
    Console.WriteLine("RoundGlyph.Draw(), "
      + "radius = {0} thickness = {1}",
      + radius, thickness);
  }
}

public class PolyConstructors {
  public static void Main() {
    new RoundGlyph(5);
  }
} ///:~
```

In **Glyph**, the **Draw( )** method is **abstract**, so it is designed to be overridden. Indeed, you are forced to override it in **RoundGlyph**. But the **Glyph** constructor calls this method, and the call ends up in **RoundGlyph.Draw( )**, which would seem to be the intent. But look at the output:

```
Glyph() before draw()
RoundGlyph.Draw(), radius = 1 thickness = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5 thickness = 2
```

When **Glyph**'s constructor calls **Draw( )**, the values of **radius** are set to their default values, not their post-construction intended values.

A good guideline for constructors is, "If possible, initialize member variables directly. Do as little as possible in a constructor to set the object into a good state, and if you can possibly avoid it, don't call any methods." The only safe methods to call inside a constructor are non-virtual.

Designing with inheritance

Once you learn about polymorphism, it can seem that everything ought to be inherited because polymorphism is such a clever tool. This can burden your designs; in fact if you choose inheritance first when you're using an existing class to make a new class, things can become needlessly complicated.

A better approach is to choose composition first, when it's not obvious which one you should use. Composition does not force a design into an inheritance hierarchy. But composition is also more flexible since it's possible to dynamically choose a type (and thus behavior) when using composition, whereas inheritance requires an exact type to be known at compile-time. The following example illustrates this:

```
//:c08:Transmogrify.cs
// Dynamically changing the behavior of
// an object via composition.
using System;

abstract class Actor {
  public abstract void Act();
}

class HappyActor : Actor {
  public override void Act() {
```

```
    Console.WriteLine("HappyActor");
  }
}

class SadActor : Actor {
  public override void Act() {
    Console.WriteLine("SadActor");
  }
}

class Stage {
  Actor a = new HappyActor();
  internal void Change() { a = new SadActor();}
  internal void Go() { a.Act();}
}

public class Transmogrify {
  public static void Main() {
    Stage s = new Stage();
    s.Go(); // Prints "HappyActor"
    s.Change();
    s.Go(); // Prints "SadActor"
  }
} ///:~
```

A **Stage** object contains a reference to an **Actor**, which is initialized to a **HappyActor** object. This means **Go( )** produces a particular behavior. But since a reference can be re-bound to a different object at run-time, a reference for a **SadActor** object can be substituted in **a** and then the behavior produced by **Go( )** changes. Thus you gain dynamic flexibility at run-time. (This is also called the *State Pattern*.) In contrast, you can't decide to inherit differently at run-time; that must be completely determined at compile-time.

A general guideline is "Use inheritance to express differences in behavior, and fields to express variations in state." In the above example, both are used: two different classes are inherited to express the difference in the **Act( )** method, and **Stage** uses composition to allow its state to be changed. In this case, that change in state happens to produce a change in behavior.

Pure inheritance vs. extension

When studying inheritance, it would seem that the cleanest way to create an inheritance hierarchy is to take the "pure" approach. That is, only methods that

have been established in the base class or **interface** are to be overridden in the derived class, as seen in this diagram:

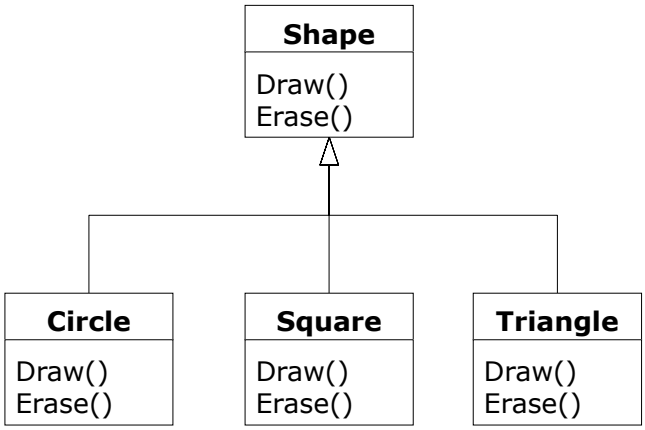


Figure 8-5: A "pure" is-a relationship

This can be termed a pure "is-a" relationship because the interface of a class establishes what it is. Inheritance guarantees that any derived class will have the interface of the base class and nothing less. If you follow the above diagram, derived classes will also have *no more* than the base class interface.

This can be thought of as *pure substitution*, because derived class objects can be perfectly substituted for the base class, and you never need to know any extra information about the subclasses when you're using them:

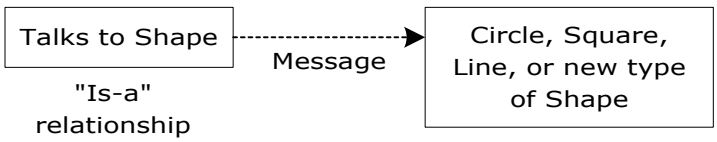


Figure 8-6: After creation, the concrete subtype is of no concern

That is, the base class can receive any message you can send to the derived class because the two have exactly the same interface. All you need to do is upcast from the derived class and never look back to see what exact type of object you're dealing with. Everything is handled through polymorphism.

When you see it this way, it seems like a pure "is-a" relationship is the only sensible way to do things, and any other design indicates muddled thinking and is by definition broken. This too is a trap. As soon as you start thinking this way,

you'll turn around and discover that extending the interface is the perfect solution to a particular problem. This could be termed an "is-like-a" relationship because the derived class is *like* the base class—it has the same fundamental interface—but it has other features that require additional methods to implement:

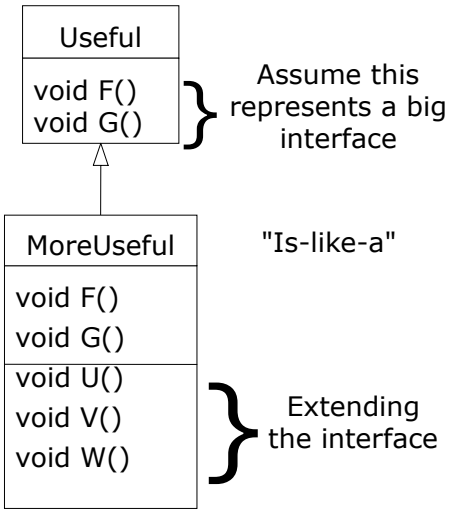


Figure 8-7: The subtype extends the base class

While this is also a useful and sensible approach (depending on the situation) it has a drawback. The extended part of the interface in the derived class is not available from the base class, so once you upcast you can't call the new methods:

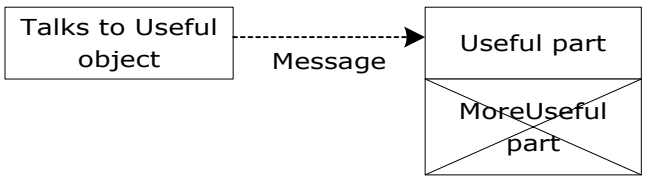


Figure 8-8: If you upcast, you can't use the extended interface

If you're not upcasting in this case, it won't bother you, but often you'll get into a situation in which you need to rediscover the exact type of the object so you can access the extended methods of that type. The following section shows how this is done.

Downcasting and run-time type identification

Since you lose the specific type information via an *upcast* (moving up the inheritance hierarchy), it makes sense that to retrieve the type information—that is, to move back down the inheritance hierarchy—you use a *downcast*. However, you know an upcast is always safe; the base class cannot have a bigger interface than the derived class, therefore every message you send through the base class interface is guaranteed to be accepted. But with a downcast, you don't really know that a shape (for example) is actually a circle. It could instead be a triangle or square or some other type.

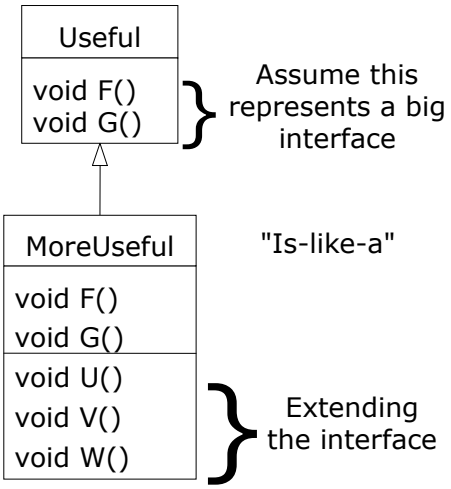


Figure 8-9: In non-"pure" interface inheritance, downcasting may be necessary

To solve this problem there must be some way to guarantee that a downcast is correct, so you won't accidentally cast to the wrong type and then send a message that the object can't accept. This would be quite unsafe. In some languages (like C++) you must perform a special operation in order to get a type-safe downcast, but in C# *every cast* is checked!

C# supports two types of downcast: a parenthesized cast that looks similar to the casts in other C-derived languages

```
MoreUseful downCastObject = (MoreUseful) myUsefulHandle;
```

and the **as** keyword

```
MoreUseful downCastObject = myUsefulHandle as MoreUseful;
```

At run-time, both these casts are checked to ensure that the **myUsefulHandle** does in fact refer to an instance of type **MoreUseful.** If this a bad assumption, the parenthesized cast will throw an **InvalidCastException** and the **as** cast will assign **downCastObject** the value of null.

This act of checking types at run-time is called *run-time type identification* (RTTI). The following example demonstrates the behavior of RTTI:

```
//:c08:RTTI.cs
// Downcasting & Run-time Type
// Identification (RTTI).

class Useful {
  public virtual void F() {}
  public virtual void G() {}
}

class MoreUseful : Useful {
  public override void F() {}
  public override void G() {}
  public void U() {}
  public void V() {}
  public void W() {}
}

public class RTTI {
  public static void Main() {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].F();
    x[1].G();
    // Compile-time: method not found in Useful:
    //! x[1].U();
    ((MoreUseful)x[1]).U(); // Parenthesized downcast
    (x[1] as MoreUseful).U(); //as keyword
    ((MoreUseful)x[0]).U(); // Exception thrown
  }
} ///:~
```

When you run this program, you will see something we've not yet discussed, Visual Studio's Just-In-Time Debugging dialog:

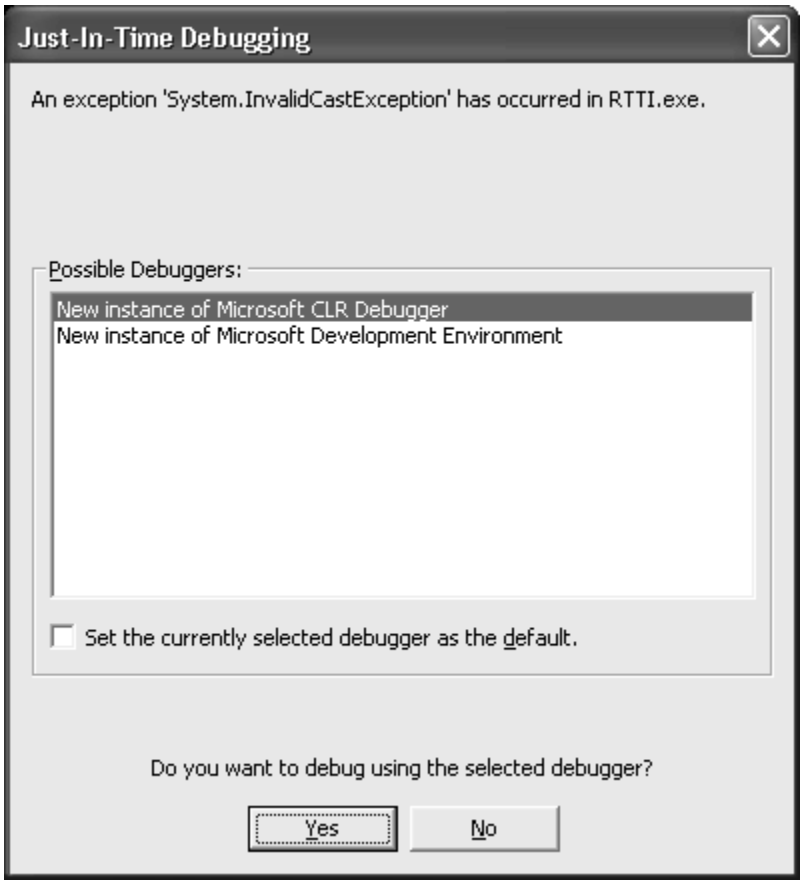

Just-In-Time Debugging
An exception 'System.InvalidCastException' has occurred in RTTI.exe.
Possible Debuggers:
New instance of Microsoft CLR Debugger
New instance of Microsoft Development Environment
Set the currently selected debugger as the default.
Do you want to debug using the selected debugger?
Yes
No

Figure 8-10: The Just-In-Time Debugging dialog box

This is certainly more welcome than a Dr. Watson dump or a Blue Screen of Death, but we know the cause – we're trying to treat **x[0]** as a **MoreUseful** when it's only a **Useful**. Select "No" and the program will end with a complaint about an unhandled **InvalidCastException**.

As in the diagram, **MoreUseful** extends the interface of **Useful**. But since it's inherited, it can also be upcast to a **Useful**. You can see this happening in the initialization of the array **x** in **Main( )**. Since both objects in the array are of class **Useful**, you can send the **F( )** and **G( )** methods to both, and if you try to call **U( )** (which exists only in **MoreUseful**) you'll get a compile-time error message.

If you want to access the extended interface of a **MoreUseful** object, you can try to downcast. If it's the correct type, it will be successful. Otherwise, you'll get an **InvalidCastException**.

There's more to RTTI than a simple cast. The **is** keyword allows you check the type of an object before attempting a downcast.

```
//:c08:RTTI2.cs
// Downcasting & Run-time Type
// Identification (RTTI).

class Useful {
  public virtual void F() {}
  public virtual void G() {}
}

class MoreUseful : Useful {
  public override void F() {}
  public override void G() {}
  public void U() {}
  public void V() {}
  public void W() {}
}

public class RTTI2 {
  public static void Main() {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].F();
    x[1].G();

    foreach(Useful u in x){
      if (u is MoreUseful) {
        ((MoreUseful) u).U();
      }
    }
  }
} ///:~
```

This program runs to completion without any exceptions being thrown. Everything stays the same except the final iteration over the **x** array. The **foreach** loop iterates over the array (all two elements of it!), but we guard the downcast with a Boolean test to ensure that we only attempt the downcast on objects of type **MoreUseful**.

Interfaces

The **interface** keyword takes the **abstract** concept one step further. You could think of it as a "pure" **abstract** class. It allows the creator to establish the form for a class: method and property names, argument lists, and return types, but no method bodies. An **interface** provides only a form, but no implementation.

An **interface** says: "This is what all classes that *implement* this particular interface will look like." Thus, any code that uses a particular **interface** knows what methods might be called for that **interface**, and that's all. So the **interface** is used to establish a "protocol" between classes. (Some object-oriented programming languages have a keyword called *protocol* to do the same thing.)

To create an **interface**, use the **interface** keyword instead of the **class** keyword. Like a class, you can add visibility modifiers such as the **public** keyword before the **interface** keyword or leave it off to give internal status so that it is only usable within the same assembly.

To make a class that conforms to a particular **interface** (or group of **interface**s), you use the colon (:) operator, just as you would to specify you were inheriting from a **Class**.

Once you've implemented an **interface**, that implementation becomes an ordinary class that can be extended in the regular way.

All interface methods are inherently **public** and **virtual**. You cannot use visibility modifiers or the **virtual** keyword in your declarations.

You can see all this in the modified version of the **Instrument** example. Note that every method in the **interface** is strictly a declaration, which is the only thing the compiler allows. In addition, none of the methods in **Instrument** are declared as either **virtual** or **public**, but they're automatically **public virtual** anyway:

```
//:c08:Music5.cs
// Interfaces.
using System;
```

```
interface Instrument {
  // Compile-time constant:
  // Cannot have method definitions:
  void Play(); // Automatically public & virtual
  string What();
  void Adjust();
}

class Wind : Instrument {
  public virtual void Play() {
    Console.WriteLine("Wind.Play()");
  }
  public virtual string What() { return "Wind";}
  public virtual void Adjust() {}
}

class Percussion : Instrument {
  public virtual void Play() {
    Console.WriteLine("Percussion.Play()");
  }
  public virtual string What() { return "Percussion";}
  public virtual void Adjust() {}
}

class Stringed : Instrument {
  public virtual void Play() {
    Console.WriteLine("Stringed.Play()");
  }
  public virtual string What() { return "stringed";}
  public virtual void Adjust() {}
}

class Brass : Wind {
  public override void Play() {
    Console.WriteLine("Brass.Play()");
  }
  public override void Adjust() {
    Console.WriteLine("Brass.Adjust()");
  }
}
```

```
class Woodwind : Wind {
  public override void Play() {
    Console.WriteLine("Woodwind.Play()");
  }
  public override string What() { return "Woodwind";}
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void Tune(Instrument i) {
    // ...
    i.Play();
  }
  static void TuneAll(Instrument[] e) {
    for (int i = 0; i < e.Length; i++)
      Tune(e[i]);
  }
  public static void Main() {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    TuneAll(orchestra);
  }
} //:~~
```

The rest of the code works the same. It doesn't matter if you are upcasting to a "regular" class called **Instrument**, an **abstract** class called **Instrument**, or to an **interface** called **Instrument**. The behavior is the same. In fact, you can see in the **Tune( )** method that there isn't any evidence about whether **Instrument** is a "regular" class, an **abstract** class, or an **interface**. This is the intent: Each approach gives the programmer different control over the way objects are created and used.

"Multiple inheritance" in C#

The **interface** isn't simply a "more pure" form of **abstract** class. It has a higher purpose than that. Because an **interface** has no implementation at all—that is, there is no storage associated with an **interface**—there's nothing to prevent many **interface**s from being combined. This is valuable because there are times when you need to say "An **x** is an **a** *and* a **b** *and* a **c**." In C++, this act of combining multiple class interfaces is called *multiple inheritance*, and it carries some rather sticky baggage because each class can have an implementation. In C#, you can perform the same act, but only one of the classes can have an implementation, so the problems seen in C++ do not occur with C# when combining multiple interfaces:

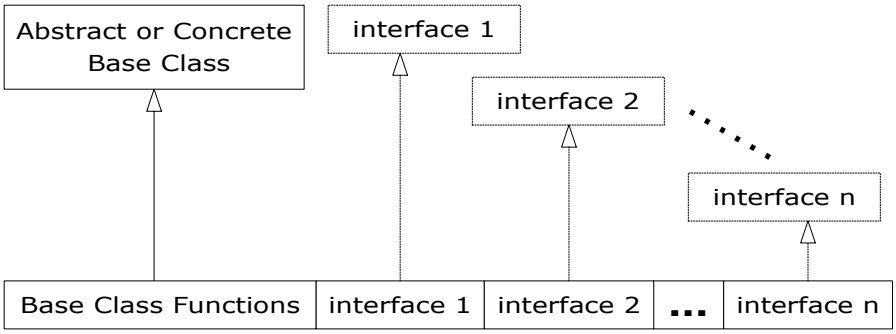


Figure 8-11: "Multiple inheritance" with interfaces

In a derived class, you aren't forced to have a base class that is either an **abstract** or "concrete" (one with no **abstract** methods). If you *do* inherit from a non-**interface**, you can inherit from only one. All the rest of the base elements must be **interface**s. You separate the base class (if there is one) and the interfaces with commas. You can have as many **interface**s as you want—each one becomes an independent type that you can upcast to. The following example shows a concrete class combined with several **interface**s to produce a new class:

```
//:c08:Adventure.cs
// Multiple interfaces.
using System;

interface ICanFight {
  void Fight();
}

interface ICanSwim {
```

```
  void Swim();
}

interface ICanFly {
  void Fly();
}

class ActionCharacter {
  public void Fight(){Console.WriteLine("Fighting");}
}

class Hero : ActionCharacter, ICanFight, ICanSwim, ICanFly
{
  public void Swim() { Console.WriteLine("Swimming");}
  public void Fly() { Console.WriteLine("Flying");}
}

public class Adventure {
  static void T(ICanFight x) { x.Fight();}
  static void U(ICanSwim x) { x.Swim();}
  static void V(ICanFly x) { x.Fly();}
  static void W(ActionCharacter x) { x.Fight();}
  public static void Main() {
    Hero h = new Hero();
    T(h); // Treat it as an ICanFight
    U(h); // Treat it as an ICanSwim
    V(h); // Treat it as an ICanFly
    W(h); // Treat it as an ActionCharacter
  }
} ///:~
```

You can see that **Hero** combines the concrete class **ActionCharacter** with the interfaces **ICanFight**, **ICanSwim**, and **ICanFly**. When you combine a concrete class with interfaces this way, the concrete class must come first, then the interfaces. (The compiler gives an error otherwise.)

Note that the signature for **Fight( )** is the same in the **interface ICanFight** and the class **ActionCharacter**, and that **Fight( )** is *not* provided with a definition in **Hero**. The rule for an **interface** is that you can inherit from it (as you will see shortly), but then you've got another **interface**. If you want to create an object of the new type, it must be a class with all definitions provided. Even though **Hero** does not explicitly provide a definition for **Fight( )**, the definition comes along

with **ActionCharacter** so it is automatically provided and it's possible to create objects of **Hero**.

In class **Adventure**, you can see that there are four methods that take as arguments the various interfaces and the concrete class. When a **Hero** object is created, it can be passed to any of these methods, which means it is being upcast to each **interface** in turn. Because of the way interfaces are designed in C#, this works without a hitch and without any particular effort on the part of the programmer.

Keep in mind that the core reason for interfaces is shown in the above example: to be able to upcast to more than one base type. However, a second reason for using interfaces is the same as using an **abstract** base class: to prevent the client programmer from making an object of this class and to establish that it is only an interface. This brings up a question: Should you use an **interface** or an **abstract** class? An **interface** gives you the benefits of an **abstract** class *and* the benefits of an **interface**, so if it's possible to create your base class without any method definitions or member variables you should always prefer **interface**s to **abstract** classes. In fact, if you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change to an **abstract** class, or if necessary a concrete class.

Name collisions when combining interfaces

You can encounter a small pitfall when implementing multiple interfaces. In the above example, both **ICanFight** and **ActionCharacter** have an identical **void Fight( )** method. This is no problem because the method is identical in both cases, but what if it's not? Here's an example:

```
//:c08:InterfaceCollision.cs
interface I1 { void F();}
interface I2 { int F(int i);}
interface I3 { int F();}
class C {
  public virtual int F() { return 1;}
}

class C2 : I1, I2 {
  public void F() {}
  public int F(int i) { return 1;}
}
```

```
class C3 : C, I2 {
  public int F(int i) { return 1;}
}

class C4 : C, I3 {
  // Identical, no problem:
  public override int F() { return 1;}
}

class C5 : C , I1 {
  public override int F(){ return 1;}
  void F(){}
}

interface I4 : I1, I3 {}

class C6: I4 {
  void F() {}
  int F() { return 1;}
}///:~
```

The difficulty occurs because overriding, implementation, and overloading get unpleasantly mixed together, and overloaded functions cannot differ only by return type. If you attempt to compile this example, the error messages say it all:

```
InterfaceCollision.cs(23,10): error CS0111: Class 'C5'
already defines a member called 'F' with the same parameter
types
InterfaceCollision.cs(22,25): (Location of symbol related
to previous error)
InterfaceCollision.cs(30,8): error CS0111: Class 'C6'
already defines a member called 'F' with the same parameter
types
InterfaceCollision.cs(29,9): (Location of symbol related to
previous error)
```

Using the same method names in different interfaces that are intended to be combined generally causes confusion in the readability of the code, as well. Strive to avoid it.

However, if it is impossible to avoid, you can prepend the name of the class or interface to the appropriate method. Thus, you could change the above to:

```
class C5 : C , I1 {
    public override int F(){ return 1; }
    void I1.F(){  }
}

class C6: I4{
   void I1.F() { }
   int I3.F() { return 1; }

}
```

In **C5**, the first **F** overrides the virtual method defined in **C**, while the second declaration, **I1.F( )** overrides the **F( )** declared in **interface I1**. **C6** similarly overrides both **I1** and **I3** declarations of method **F( )**.

Extending an interface with inheritance

You can easily add new method declarations to an **interface** using inheritance, and you can also combine several **interface**s into a new **interface** with inheritance. In both cases you get a new **interface**, as seen in this example:

```
//:c08:HorrorShow.cs
// Extending an interface with inheritance.

interface Monster {
  void Menace();
}

interface DangerousMonster : Monster {
  void Destroy();
}

interface Lethal {
  void Kill();
}

class DragonZilla : DangerousMonster {
  public void Menace() {}
  public void Destroy() {}
}
```

```
interface Vampire  : DangerousMonster, Lethal {
  void DrinkBlood();
}

public class HorrorShow {
  static void U(Monster b) { b.Menace();}
  static void V(DangerousMonster d) {
    d.Menace();
    d.Destroy();
  }
  public static void Main() {
    DragonZilla if2 = new DragonZilla();
    U(if2);
    V(if2);
  }
} ///:~
```

DangerousMonster is a simple extension to **Monster** that produces a new **interface**. This is implemented in **DragonZilla**.

Summary

Polymorphism means "different forms." In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the dynamically bound methods.

You've seen in this chapter that it's impossible to understand, or even create, an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like a **switch** statement can, for example), but instead works only in concert, as part of a "big picture" of class relationships. People are often confused by other, non-object-oriented features of C#, like method overloading, which are sometimes presented as object-oriented. Don't be fooled: If it isn't late binding, it isn't polymorphism.

To use polymorphism—and thus object-oriented techniques—effectively in your programs you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other. Although this requires significant effort, it's a worthy struggle, because the results are faster program development, better code organization, extensible programs, and easier code maintenance.

Interfaces contribute enormously to good designs. By explicitly separating the concept of "What needs to be done" from "How it is implemented," designers are

forced to consider the possibility of a difference between the two. Even if there isn't an immediately obvious difference, the complex types that form the core of an application will benefit from having their interfaces made explicit. There are several testing and debugging techniques that involve inserting classes "between" an interface and its implementation. Over time, you'll become better at recognizing situations where an interface, an abstract class, or a concrete base class is appropriate. But at this point in this book you should at least be comfortable with the syntax and semantics. As you see these language features in use you'll eventually internalize them.

Exercises

1. Create an **interface** containing three methods, in its own namespace. Implement the interface in a different namespace.

2. Prove that all the methods in an **interface** are automatically **public**.

3. In **Sandwich.cs**, create an interface called **FastFood** (with appropriate methods) and change **Sandwich** so that it also implements **FastFood**.

4. Create three **interface**s, each with two methods. Inherit a new **interface** from the three, adding a new method. Create a class by implementing the new **interface** and also inheriting from a concrete class. Now write four methods, each of which takes one of the four **interface**s as an argument. In **Main( )**, create an object of your class and pass it to each of the methods.

5. Modify Exercise 4 by creating an **abstract** class and inheriting that into the derived class.

6. Modify **Music5.cs** by adding a **Playable interface**. Move the **Play( )** declaration from **Instrument** to **Playable**. Add **Playable** to the derived classes by including it in the **implements** list. Change **tune( )** so that it takes a **Playable** instead of an **Instrument**.

7. In **Adventure.cs**, add an **interface** called **CanClimb**, following the form of the other interfaces.

8. Choose a program in your robotic party servant system with which you are not satisfied. Describe the value you are trying to achieve in terms of an interface (**IMixCocktail** or **IDiscJockey** or what-have-you). Change your existing class so that it implements this interface, but change your

existing implementation as little as possible. Write your **Main( )** so that it upcasts your implementation to the interface.

Write a *new* class that also implements the interface, but this time with your desired improvements. Write your **Main( )** to use this new class. The change in **Main( )** should only require you to change the right-hand side of one line of code:

```
IPourDrinks ipd = new OriginalImplementation();
```

becomes:

```
IPourDrinks ipd = new ImprovedImplementation();
```

9. Implement this interface in at least 3 different ways:

```
interface IReverseString{
   ///<summary>Returns the input string with
   ///its characters reversed</summary>
   string Reverse(string input);
}
```

Write a test program that allows you to choose the different implementations, upcasts to **IReverseString**, and uses **DateTime.Now** to time their relative performance.

10. "Simulation games" such as The Sims™ work by assigning to each element in the game a large number of characteristics, some of which interact and some of which don't. For instance, in a game about managing a zoo, you might have people spend more time in front of big animals such as elephants and tigers, while another aspect of the game might involve preparing food for herbivores versus carnivores, which wouldn't have any influence on viewing. Identify some of the characteristics that might be interesting to simulate in such a zoo management game. A driving requirement is that level designers should be able to create a new animal or situation by specifying artwork, assemblies of pre-existing code, and a minimum of custom coding. Identify interfaces and abstract classes to help achieve this goal.

11. Using your work from exercise 10, implement at least 3 animals ("implement" in this case meaning "write a description of the desired action or property to the console").

12. Exercise 11 required you to refine your exercise 10 work, modifying, adding, and perhaps deleting interfaces, methods, and characteristics.

Consider the challenges inherent in repeating this process of "design-a-little, implement-a-little" over the course of a product lifespan of, say, five years. Write a 500-word argument defending or refuting the proposition that "Tools to help produce, store, and track design artifacts such as class diagrams and documentation are worth a serious investment in time and effort."

9: Coupling and Cohesion

Data encapsulation, inheritance, and polymorphism are the cornerstone capabilities of object orientation, but their use does not automatically create good or even passable design. Good software design arises from dividing a problem into coherent parts and tackling those parts independently, in a manner that's easy to test and change. Object orientation facilitates just such partitioning of concerns, but requires you to understand the forces that make some software designs better than others. These forces, coupling and cohesion, are universal to all software systems and provide a basis for comparing software architectures and designs.

When we speak of "architecture" and "design" in software systems, we can be speaking of many different things – the physical structure of the network, the set of operating system and server providers chosen, the graphic design of the entire client-facing Website, the user-interface design of applications we write, or the internal structure of the programs being written. All of these factors are important and it's a shame that we have not yet developed a common vocabulary for giving each of them their just attention. For the purposes of this chapter, we are solely concerned with the internal structure of programs – decisions that will be made and embodied in the C# source code you write.

Software as architecture vs. software architecture

As often as not, the lead programmer in a team has the title of Software Architect. The popularity of this title comes from the popular view that the challenges of building a software system are similar to the challenges of building a skyscraper or bridge.

The view of software *as* architecture stems from the undeniable truth that software often fails. Most of those who have studied larger software projects (projects in excess of 5,000 function points, which translates to probably 175,000 lines of C#) have concluded that aspects other than code construction are the most important drivers of success or failure. What these people see in large projects, time and again, are failures relating to requirements and failures relating to integration: teams build the wrong thing and then they can't get the pieces they build to fit together. Based on these observations, the challenges of software seemed to parallel the challenges of architecture – the craftsmanship of the individual worker is all well and good, but success comes from careful planning and coordination.

Every few years, the pendulum of popular opinion swings from this view towards a view that emphasizes the individual contributions of talented programmers. A few years later, the pendulum predictably swings the other way. This view of software holds that, like it or not, code defines the software's behavior and that diagrams and specifications do not generally capture the real issues at hand when designing programs. Proponents of this view argue that the software *as* architecture view is primarily pushed by consultants and vendors selling expensive tools that generate a lot of paper but little product.

Until recently, the escalation of defect costs over time was the trump card for the software *as* architecture advocates. In 1987, Barry Boehm published a paper in which he established that a defect corrected for $1 in the requirements phase could cost $100 to fix once deployed[1]. With those economics, big up-front efforts were obviously worthwhile. It's doubtful that such numbers have much meaning today; Boehm himself concluded in 2001 that for "smaller, noncritical software systems" the cost escalation was "more like 5:1 than 100:1." The explosively popular set of practices called Extreme Programming is based on "the technical premise [that] the cost of change [can rise] much more slowly, eventually reaching an asymptote."[2]

So should you view software development as an undertaking akin to building a bridge or as one akin to growing a garden? Well, there are a few things we can say for sure about software development: Programmers always overestimate their long-term productivity. Most large software projects still cost more than expected. Most projects still take longer to finish than expected. Many are

---

[1] *Software Defect Reduction Top 10 List*, Boehm and Basili, IEEE Computer 34(1).

[2] *Extreme Programming Explained*, pg. 23.

cancelled prior to completion and many, once deployed, fail to achieve the business goals they were intended to serve. Beyond these generally pessimistic statements, the variances between individuals, teams, companies, and industry sectors swamp the statistics. Useful software systems range from programs that are a few hundred lines long (the source code to **ping** is shorter than some of the samples in this book!) to programs in excess of a million lines of code. Teams can range from individuals to hundreds. A successful program may be a function-optimizer that, on a single run out of a thousand, produces a meaningful result or a life support system that, literally, never fails (yes, provably correct software is possible). So when people start spouting statistics or pronouncements about "best practices," at best they're over-generalizing and at worst they're trying to scare you into buying whatever it is they're selling[3].

We do know that there's at least one major flaw in the software as architecture metaphor, and that is that unlike a building that remains under construction until a designated "grand opening," software should be built and deployed incrementally – if not one "room at a time" then at least "one floor at a time." This is, of course, easier in some situations than in others, but one of the great truisms of software development management is that from the first weeks of the project, you should be able to ship, at least potentially, your latest build.

What is software architecture?

Every non-trivial program should have an overall ordering principle. While that's vague enough to allow for a lot of different interpretations of what does and does not constitute an architecture, you don't gain much by viewing .NET as your systems' architecture. Rather, you should consider the way that data flows in and out of your system and the ways in which it is transformed. At such a level of abstraction, there are fewer variations in structure than you might guess.

Most programs are going to have at least a few subsystems, and each may have its own architecture. For instance, almost all systems require some kind of user interface, even if it's just for administration, and UI subsystems are typically architected to follow either the Model-View-Controller pattern or the Presentation-Abstraction-Control pattern (both of which are thoroughly explained in Chapter 14). However, neither of these architectures speaks to how

---

[3] Be especially dubious if you hear "70% failure rate." This, like the $300B that Y2K failures were going to cost, is a number that originated in a single profit-motivated study but the number has been repeated so many times that it's taken on a life of its own.

the program should structure the creation of business value (what goes on in the "Model" or "Abstraction" parts of MVC and PAC respectively).

Simulation architectures: always taught, rarely used

Many people are taught that object oriented systems should be structured to conform to objects in the problem domain, an architectural pattern we could call Simulation. There is certainly historical precedent for this: object oriented programming was in fact invented in response to the needs of simulation programming. Unfortunately, outside of video games and certain types of modeling, Simulation is not generally a useful architecture. Why? Well, because the business interests of most companies are usually focused on transactions driven by non-deterministic forces (i.e., people) and it just doesn't do much for the bottom line to try to recreate those forces inside the computer system. Amazon's ordering system probably has a **Book** class, and maybe an **Author** class, but I'll wager it doesn't have an **Oprah** class, no matter how much the talk-show host's recommendations influence purchasers.

Simulation architectures tend to exemplify the concept of cohesion, though. Cohesion is the amount of consistency and completion in a class or namespace, the amount by which a class or namespace "hangs together." In their proper domain, simulations tend to have great cohesion: when you've got a **Plant** object and an **Animal** object, it's pretty easy to figure out where to put your **Photosynthesis( )** method. Similarly, with classes derived from real world nouns, one doesn't often have "coincidental cohesion" where a class contains multiple, completely unrelated interfaces.

Client/server and *n*-tier architectures

When corporate networks first became ubiquitous in the early 1990s, there was a great deal of excitement about the possibility of exploiting the desktop computers' CPU power and display characteristics, while maintaining data integrity and security on more controllable servers. This client/server architecture quickly fell out of favor as client software turned out to be costly to develop and difficult to keep in sync with changing business rules. Client/server was replaced with 3-tier and *n*-tier architectures, which divide the world into 3 concerns: a presentation layer at the client, one or more business rules layers that "run anywhere" (but usually on a server), and a persistence or database layer that definitely runs on a server, but perhaps not the same machines that run the business rules layers.

Originally popular for corporate development, the dot-com explosion entrenched the *n*-Tier architecture as the most common system architecture in development today. Web Services, which we'll discuss in more detail in Chapter 18, are *n*-tier architectures where the communication between layers is done with XML over Web protocols such as SOAP, HTTP, and SMTP.

In this architecture, all the interesting stuff happens in the business-rule tiers, which are typically architected in a use-case-centric manner. "Control" classes represent the creation of business value, which is usually defined in terms of a use-case, an atomic use of the system that delivers value to the customer. These control objects interact with objects that represent enduring business themes (such as Sales, Service, and Loyalty), financial transactions (such as Invoices, Accounts Receivable and Payable, and Taxes), regulatory constraints, and business strategies. Classes that map into nouns in the real world are few and far between.

The theme of such business-tier architectures is "isolating the vectors of change." The goal is to create classes based not on a mapping to real-world nouns but to patterns of behavior or strategy that are likely to change over time. For instance, a retail business is likely to have some concept of a Stock Keeping Unit – a product in a particular configuration with a certain manufacturer, description, and so forth – but may have many different strategies for pricing. In a typical business-tier architecture, this might lead to the creation of classes such as **StockKeepingUnit**, **Manufacturer** , and an interface for **PricingStrategy** implemented by **EverydayPrice, SalePrice,** or **ClearancePrice**. The "vector of change" in this case is the **PricingStrategy**, which is manipulated by a control object; for instance, an object which lowers prices when a **StockKeepingUnit** is discontinued:

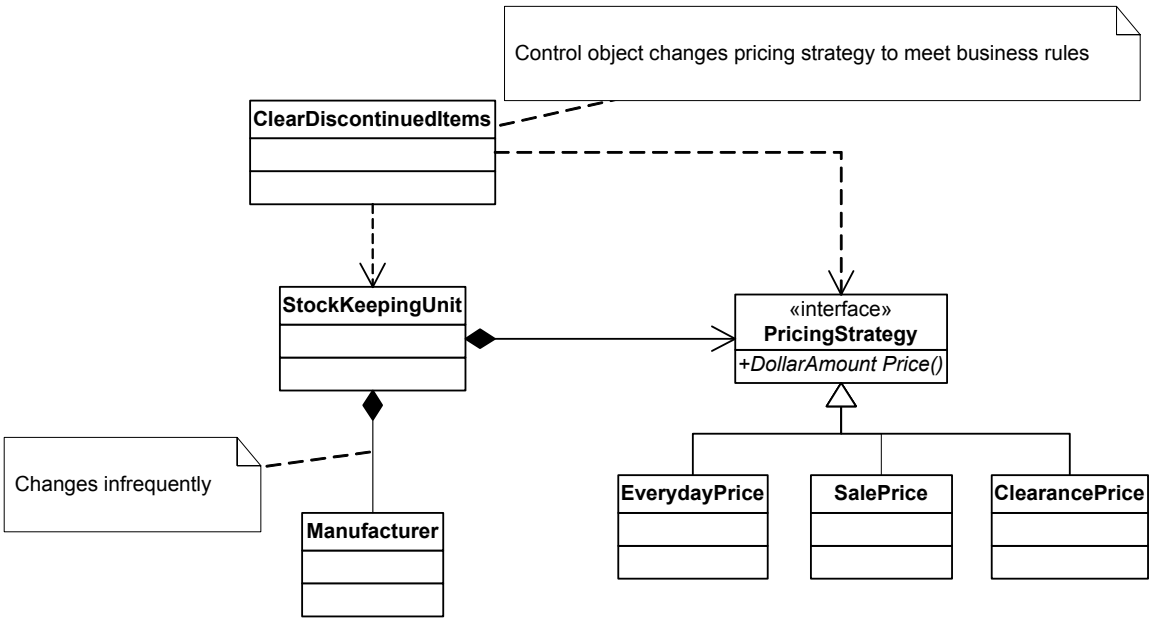


Figure 9-1: Not all objects correspond to "physical things in the real world."

This design fragment is typical of the types of structures one sees in business-tier architectures: use-cases are reified (a fancy word for "instantiated as objects"), there is extensive use of the Strategy pattern, and there are lots of classes which do not map to tangible objects in the real world.

The business rule "When a product is discontinued by its manufacturer, place it on sale until we have less than a dozen in inventory, then price it for clearance," might look like this in code:

```
//:c09:Manufacturing.cs
class ClearDiscontinuedItems {
  static readonly int MIN_BEFORE_CLEARANCE = 12;

  public void Discontinued(StockKeepingUnit[] line){
    foreach(StockKeepingUnit sku in line){
      int count = sku.NumberInStock;
      if (count < MIN_BEFORE_CLEARANCE) {
        sku.PricingStrategy =
        new ClearancePrice();
      } else {
        sku.PricingStrategy = new SalePrice();
      }
```

```
    }
  }
}
```

In a Simulation architecture, a **StockKeepingUnit** would price itself when discontinued:

```
class StockKeepingUnit {
  //...
  static int numberInStock;
  public int NumberInStock{
    get { return numberInStock; }
    set { numberInStock = value; }
  }
  static readonly int MIN_BEFORE_CLEARANCE = 12;

  PricingStrategy ps;
  internal PricingStrategy PricingStrategy{
    get { return ps; }
    set { ps = value; }
  }

  void Discontinue(){
    if (numberInStock < MIN_BEFORE_CLEARANCE) {
      this.ps = new ClearancePrice();
    } else {
      this.ps = new SalePrice();
    }
  }
}
```

As can be seen in this design fragment, business-tier architectures tend to look less cohesive than simulation architectures for small problems. In real-world applications, though, there are typically dozens or hundreds of use-cases, each of which may have several business rules associated with it. Business rules tend to involve a lot of different factors and yet are the natural unit of thought for the domain experts, so business-tier architectures tend to result in clearer lines of communication between development teams and domain experts. “We need to ask a question about the ‘price discontinued items’ business rules” is much clearer to domain experts than “We need to ask a question about how an SKU prices itself when it’s discontinued.”

Layered architectures

Once upon a time, computers were primarily used to automate repetitive calculations: tide charts, gunnery tables, census statistics. Nowadays, CPU-intensive math is one of the least frequent tasks asked of a computer. We imagine there are many younger people who don't know that, essentially, computers are nothing but binary calculators. This is because layer after layer of increasing abstraction has been placed between the end-user and the underlying hardware. We're not sure this is entirely beneficial when it comes to programming, but we certainly don't want to go back to the days when the big trick in graphics programming was doing your work when the screen was doing a vertical retrace. Layers of abstraction are among the most powerful architectures over the long term, removing entire areas from the concerns of later programmers.

Layered architectures are commonly used when implementing protocols, since protocols themselves have a tendency to be layered on top of other protocols. Layers are also ubiquitous when working with device drivers or embedded systems (not likely scenarios for C# today, but who knows what tomorrow holds?).

The three places where one can anticipate a C# team looking towards layered architectures are in the areas of object-relational mapping, concurrent libraries, and network communications. The first two are areas where the .NET framework does not provide sufficient abstraction and the last is a fast-moving area where C# has the potential to really shine.

Layered architectures always seem to be difficult to get right. In a layered architecture, all classes in a layer should share the same abstraction layer, and each layer should be able to perform all its functions by only relying on its own classes and those of the layer immediately "below" it. Pulling this off requires a more complete understanding of the problem domain than is typically available in the early stages of system development, when architecture is decided. So usually with a layered architecture, you face two possibilities: a lot of upfront effort or being very conscientious about moving objects and refactoring based on abstraction level.

The great benefits of a layered architecture come down the line when an understanding of the working of the lower levels is rarely or never needed. This is the situation today with, say, graphics, where no one but game programmers need concern themselves with the complex details of screen modes, pixel aspects, and display pages (and with modern game engines built on DirectX and the

prevalence of graphics accelerator cards, apparently even game programmers are moving away from this level of detail).

Problem-solving architectures

It's increasingly rare to develop a system that solves a problem that is otherwise intractable. This is too bad, because there are few thrills greater than developing a tool and watching that tool accomplish something beyond your capabilities. Such systems often require the development of specialized problem-solving architectures — blackboard systems, function optimizers, fuzzy logic engines, support-vector machines, and so forth. These problem solvers may themselves be architected in any number of ways, so in a sense you could say that this is a variation of the layered architecture, where the problem-solver is a foundation on which the inputs and meta-behavior rely. But to solve a problem using, say, a genetic optimizer or expert system requires an understanding of the characteristics of the problem-solver, not so much an understanding of the way that the problem-solver is implemented. So if you're fortunate enough to be working in such a domain, you should concentrate on understanding the core algorithms and seeing if there's a disconnect between the academic work on the problem-solver and the task at hand; many such projects fail because the problem-solver requires that the input or meta-data have a certain mathematical characteristic. As an object lesson, research on artificial neural networks languished for more than a decade because the original work used a discontinuous "activation function." Once someone realized the mathematical problem with that, a famous "proof" of their limitations fell by the wayside and neural nets became one of the hottest research topics in machine learning!

Dispatching architectures

An amazingly large amount of value can be derived from systems that manage the flow of information without a great deal of interaction with the underlying data. These systems typically deal with the general problem of efficiently managing the flow from multiple, unknown-at-compilation-time sources of data with multiple, unknown-at-compilation-time sinks for data. "Efficiently" is sometimes measured in terms of speed, sometimes in terms of resources, and sometimes in terms of reliability. Sometimes the connections are transient, in which case the architectures are usually called "event driven" and sometimes the connections are longer lasting, in which case multiplexing and demultiplexing architectures are commonly used.

Dispatching solutions tend to be relatively small and can sometimes be seen as design elements, not the architectural organizing principle for an entire system or

subsystem. But they often play such a crucial role in delivering value that they deserve the greatest emphasis in a system's development.

"Not really object-oriented"

One of the most common criticisms that you'll hear about a piece of software is that, although it's written with an object-oriented language, "it's not really object oriented." There are two root causes for this criticism, one of which is serious and worth criticizing, and one of which betrays a lack of understanding on the part of the criticizer.

The frivolous criticism comes from those who mistakenly believe that Simulation architectures are the only valid ordering principle for object-oriented systems. Each architecture has its strengths and weaknesses and while Simulation architectures have historical ties with object orientation, they aren't the best choice in many, maybe most, real-world development efforts. There are many more architectures than have been discussed here; we've just touched on some of the more popular ones.

The serious criticism, the thing that makes too many systems "not really object oriented" is low-level design that doesn't use object-oriented features to improve the quality of the software system. There are many different types of quality: speed of execution, memory or resource efficiency, ease of integration with other code, reusability, robustness, and extensibility (to name just a few). The most important measure of software quality, though, is the extent to which your software can fulfill business goals, which usually means the extent to which the software fulfills user needs or the speed with which you can react to newly discovered user needs. Since there's nothing you can do about the former, the conclusion is "the most important thing you can do is use object-oriented features to improve the speed with which you can react to newly discovered user needs."

This is governed by the coupling and cohesion of your code.

Coupling

Dependence describes the extent to which a type or a method relies on another to accomplish its behavior. Class **A** is said to be highly dependent on Class **B** if a change in class **B**'s implementation is very likely to cause a change in **A**'s behavior. The sample programs in this book are highly dependent on the **Console** class and its **WriteLine( )** method!

Coupling is just another word for dependence between two different methods or types. There are several types of coupling, of varying levels of trouble:

Independent coupling

When two classes or methods do not share any data, they are independent. The simple use of another class does not necessarily mean that there is a coupling to the other class; in this example, **A** and **B** are still independent:

```
//:c09:IndependentCoupling.cs
class A {
  public void Foo(){
    B b = new B();
    b.Bar();
  }
}

class B {
  public void Bar(){
    //A change here will not affect A's behavior
  }
}
```

Data coupling

When class A passes an integral value type to B, they are data-coupled. The majority of sample programs in this book are data-coupled to **Console** because they pass in a **string** value to its **WriteLine( )** method. The behavior of the sample program's would change if there was a change in either the passed-in **string** or in the implementation of **Console.WriteLine( )**.

Stamp coupling

Stamp coupling occurs when class A passes a reference type or a **struct** to B. Although stamp coupling is usually considered to be okay, it is slightly more obscure than data coupling in that the casual reader of the method call signature cannot tell exactly which fields in the parameter are used by the method.

```
//:c09:StampCoupling.cs
class A {
  public void Foo(){
    B b = new B();
    b.Bar(this);
    //this.f1 and this.f2 may be changed
```

```
  }

  int f1;
  public int Field1{
    get { return f1;}
    set { f1 = value;}
  }

  int f2;
  public int Field2{
    get { return f2;}
    set { f2 = value;}
  }
}

class B {
  public void Bar(A myA){
    //Could affect myA.Field1 or myA.Field2
  }
}
```

Tramp coupling

Tramp coupling is the name for when a parameter is passed, unchanged, through many intermediate methods before being used.

```
//:c09:TrampCoupling.cs
using System.Drawing;
class Car {
  void BuildCar(Color bodyColor){
    BuildChassis(bodyColor);
  }

  void BuildChassis(Color bodyColor){
    //...etc...
    InstallEngine(bodyColor);
  }

  void InstallEngine(Color bodyColor){
    //...etc...
    PutOnTires(bodyColor);
```

```
  }

  void PutOnTires(Color bodyColor){
    //...etc...
    PaintBody(bodyColor);
  }

  void PaintBody(Color bodyColor){
    //Finally use the bodyColor!
  }
  public static void Main(){
    Car c = new Car();
    c.BuildCar(Color.Red);
  }
}///:~
```

Tramp coupling is confusing because a method's parameters should all have importance. If a **Color** is passed to **InstallEngine( )**, the implication is that the **Engine** relies on the **Color** (it's either *stamp* or *data* coupled to the method that calls **InstallEngine**). Instead of using tramp coupling, the **Car** class should have a field that sets the desired body color when it is first calculated and **PaintBody( )** should just read the value of the field. This introduces *state* into the equation – the behavior of **Car.PaintBody( )** relies on calls that are made at another point in **Car**'s lifecyle. There's nothing wrong with state, so long as your object is never in an invalid state; constructors and field initializers give you the tools to ensure that your fields always have some reasonable default value.

Control coupling

If the logic of class B is controlled by a parameter given it by class A, the two classes are control-coupled. The crucial concept is that the class A determines the *logic* that class B will use. If class B makes *its own decision*, it is not control coupling.

```
class CalendarPrint{
  public void ControlCoupledFeb
    (DayOfWeek firstIsOn, bool isLeap){
    //etc.
  }

  public void JustDataCoupledFeb(int year){
    //Could calc day of week of the first, leap year
  }
```

```
}
```

For this Calendar-printing class to print February, it needs to know what day of the week the first falls on and whether or not the year is a leap-year. These two values could be passed in, as they are in this example's **CalendarPrint.ControlCoupledFeb( )** method. Imagine the code that is needed to call this:

```
//:c09:ControlCoupling.cs
class CalendarPrint{
  public void ControlCoupledFeb
    (DayOfWeek firstIsOn, bool isLeap){
    //etc.
  }

  public void JustDataCoupledFeb(int year){
    //Could calc day of week of the first, leap year
  }

  public void PrintJanuary(DayOfWeek beginsOn){
    //...etc...
  }
}

class CalendarMaker{
  public void PrintYear(int year){
    CalendarPrint cp = new CalendarPrint();
    DayOfWeek firstIsOn = CalcDayOfWeekForJan(year);
    cp.PrintJanuary(firstIsOn);
    firstIsOn += 3;
    bool isLeap = IsLeapYear(year);
    cp.ControlCoupledFeb(firstIsOn, isLeap);
    //...etc...
  }

  DayOfWeek CalcDayOfWeekForJan(int year){
    //...etc...
  }

  bool IsLeapYear(int year){
    //...etc...
  }
```

```
}
```

Control coupling is one of the most common design mistakes in object-oriented code. While it's not *inconceivable* that you'd have one class for printing and another for doing the date calculations, it's *almost certainly* a better design if both of these concerns were handled by a single class.

External, Common, and Content coupling

If two classes communicate via a non-native object, such as via a file, they are externally coupled. Challenges with external coupling include more failure modes (what if the file is deleted by an unknowing user?) and much harder debugging, because it is harder to trace the exact state transitions of the communication object. Internet programming frameworks often rely on external coupling via cookies or URL rewriting.

Common coupling occurs when two classes are dependent on the same static data. Obviously, any change to the static data will affect all the classes that rely on it.

When **A** and **B** directly modify each other's internal state without going through properties, they are content coupled. This is the most obvious form of coupling and most designers learn to avoid it very quickly.

Cohesion

The never-to-be-reached goal with coupling is fully independent objects and methods. Seemingly, the best way to achieve this is to place all your work within a single **Main( )** method; you'll still be dependent on the .NET Framework Library methods, but you won't have any dependencies between your own methods and objects because you won't define any! Needless to say, something else is in play in good designs: cohesion.

Cohesion is the degree to which all the elements of a method or type are related to each other. It is not quite the opposite of coupling, but it's related. In general, the more cohesive a class, the looser is its external coupling. But there is a problem: cohesion tends to break a task into more and more sub-tasks, but coordinating those sub-tasks tends to increase coupling. The fun in software design is attempting to discover a way to have your highly cohesive cake and loosely couple it, too.

A cohesive class is one that implements the intuitive set of behaviors implied by its name – a cohesive **BaseballStatistics** class would contain not just the

winning scores, but the myriad details that fans expect. The questions that are asked to increase cohesion are the same that determine whether something should be turned into an object: “Does this make sense as something whose identity should be separate from other things?” and “Do these things intuitively belong together?” These questions, not “Does this correspond to something physical in the real world?” are what should guide your object-oriented designs.

Just as there are several types of coupling, so too are there several types of cohesion.

Functional cohesion

A functionally cohesive method is one that is self-contained, whose every argument has import, and that modifies one thing.

```
int Add(int x, int y){
  return x + y;
}
```

is an example of such a method. Such methods are easy to read, test, and modify.

Sequential cohesion

Many methods use the output of one step as inputs to another step. When this happens, the methods are said to exhibit sequential cohesion (or *sequential association*).

```
class AbstractExpressionist{
  public void MakeCanvas(){
    Paint p = ChooseRandomPaint();
    SplatterPaintRandomly(p);
  }
}
```

The method **AbstractExpressionist.SplatterPaintRandomly( )** is sequentially associated with **AbstractExpressionist.ChooseRandomPaint( )**: one cannot lay down a brushstroke until one has chosen a paint. All object methods are sequentially associated with the constructor, while static methods are sequentially associated with the static constructor and class loading.

Try to avoid creating sequential associations between methods that are not private. While it’s not a great sin to require some amount of sequencing from the client programmer, strive for functional cohesion in the design of the non-private methods in your classes. Consider this poor design:

```
//:c09:CarAndDriver1.cs
//Poor design
class Driver{
  Car c = new Car();

  public void Start(){
    c.PutInNeutral();
    c.InsertKey();
    c.TurnKey();
  }
}

class Car{
  public void PutInNeutral(){ … }
  public void InsertKey(){ … }
  public void TurnKey(){ … }
  //…etc…
}///:~
```

Here, the beginning designer might think that **Driver.Start( )** is just calling the sequentially associated **Car.PutInNeutral( )**, **Car.InsertKey( )**, and **Car.TurnKey( )** methods and that because the starting sequence does not directly manipulate any of the **Car** instance data, the **Start( )** method may as well be in the **Driver** class. Wrong.

Put aside the appeal to intuition that **Car** is the "obvious" place to put the **Start( )** method and objectively look at the choice to put **Start( )** in **Driver**. One thing that jumps out immediately is that to test the **Driver.Start( )** design, you would need to create a **Driver** object and a **Car** object. Whereas, if you had:

```
class Car{
  public void PutInNeutral(){ … }
  public void InsertKey(){ … }
  public void TurnKey(){ … }
  public void Start(){
    PutInNeutral();
    InsertKey();
    TurnKey();
  }
  //…etc…
}
```

you would only need to create a **Car** object and **Start( )** would call the methods on **this** (and raise the question: should only **Start( )** should be public?). One of the key insights of the Extreme Programming movement is that if something is hard to test, there's probably something wrong with the design. Conversely, the easier something is to test, the better the design is likely to be (plus, it's *easier*).

Further, think about modifying the **Driver.Start( )** design to allow for the logic that only stick-shift cars need be put in neutral before starting:

```
//:c09:CarAndDriver2.cs
//Consequence of following the previous poor design
class Driver {
  void Start(Car c){
    if (c is StickShift) {
      StickStart((StickShift) c);
    } else {
      QuickStart(c);
    }
  }

  void StickStart(StickShift c){
    c.PutInNeutral();
    QuickStart(c);
  }

  void QuickStart(Car c){
    c.InsertKey();
    c.TurnKey();
  }
  //...etc...
  }
```

This is typical of the refactoring that a programmer with a procedural background would be likely to produce early in their object-oriented days. A person with more object-oriented experience would likely do something like the following, which uses a virtual method call[4]:

```
//:c09:CarAndDriver3.cs
abstract class Car {
```

[4] Putting aside the question of whether introducing new classes such as **Key** and **Transmission** might not be the best way to handle the issue.

```
  abstract public void Start();
  protected void QuickStart(){
    InsertKey();
    TurnKey();
  }

  public void InsertKey(){ //…etc…
  }
  public void TurnKey() { //…etc…
  }
}

class StickShift : Car {
  public override void Start(){
    PutInNeutral();
    QuickStart();
  }

  void PutInNeutral(){ //…etc…
  }
}

class Automatic : Car {
  public override void Start(){
    QuickStart();
  }
}

class Driver {
  Car c;

  void Start(){
    c.Start();
  }

  //...etc...
}///:~
```

In this example, the transmission logic is associated with the sub-type of the **Car**, not the **Driver**. This means that a **Driver** can start any kind of **Car**, and new types of transmission could be added to the mix by subtyping **Car** without touching existing code.

Communicational cohesion

Communicational association occurs when a group of methods acts on the same, single piece of data. This is the part-and-parcel of object-oriented programming: instance methods work on the data associated with the **this** reference, while static methods work on the unique class data. Consider the **String** class, for instance, and all of its methods for converting, trimming, searching, and concatenating: all sorts of behavior are available to work on the particular string referenced by the particular variable.

Communicational association and sequential association are different in that sequential association is an assembly line, while the order in which communicationally associated methods are called is unimportant.

Of course, the **this** object is only a stepping-stone to individual instance fields; if you had, for instance:

```
//:c09:Boombox.cs
class Boombox{
  float fmRadioFrequency;
  void TuneInRadio(float frequency){
    fmRadioFrequency = frequency;
  }

  int currentCdTrack;
  void SetCdTrack(int track){
    currentCdTrack = track;
  }
}///:~
```

it would be incorrect to say that **Boombox.TuneInRadio( )** and **Boombox.SetCdTrack( )** exhibited communicational cohesion. (Note also that these methods would be better implemented as properties, which would also not be considered communicationally cohesive.)

Although the general rule of object-oriented programming is to place both the data and all the behavior associated with manipulating that data inside a single class (in our example, placing the **Start( )** method within the **Car** class rather than the **Driver** class), there are times when you are *trying* to make a new, independent identity for the *logic* that is separate from the identity of the *data*.

This will be discussed in more detail later in this chapter, when we discuss multi-tiered architectures.

Procedural cohesion

The original **Driver.Start( )** method that called a series of methods on the **Car c** was marginal, but things are pretty clearly awry by the time methods are using *procedural association*, in which a method embodies sequential logic on more than one object that is not **this**.

```
//:c09:BadDriver.cs
class Driver {
  Car c;
  Cellphone p;

  void Start(){
    c.PutInNeutral();
    c.InsertKey();
    c.TurnKey();

    p.PlugIn();
    p.HandsFree = true;
    if (DestinationHome()) {
      p.Dial(p.SpeedDial.Home);
      //...etc...
    }
  }
}///:~
```

Procedurally associated methods are hard to test and therefore hard to modify. This version of **Driver.Start( )** is sequentially controlling method calls in both the **Car** and the **Cellphone**; to test this method, you'd have to fully exercise all variations on the **Car** starting procedure and on the **Cellphone** initialization. Clearly, you'd want to move the logic for these responsibilities into the **Car** and **Cellphone** classes.

Temporal cohesion

Temporal association occurs when a method's behavior is related to the time at which it is called. The most common example of this might be a threaded program that "wakes up" periodically and performs some series of tasks related only by the fact they're done at a particular time (check for new email, defragment the hard drive, send the day's credit card transactions to the bank, etc.). Batch-mode processing, where temporal association is most likely to crop

up, is fairly rare now that we have ample computing resources at hand. Additionally, in C# individual threads are relatively cheap and easy to use; there are no obvious temptations that would lead to temporal association. Threading will be covered in detail in chapter 16.

Logical cohesion

Logical association occurs when a method's logic is determined by an external method that passes in a *control value*.

```
//:c09:LogicalAssociation.cs
using System;

class Reservation{
  static ConfirmationNumber ConfirmReservation
  (string name, DateTime date, bool createIfNeeded){
    Reservation res = null;
    if (date == DateTime.Today) {
      res = CheckTodaysArrivals(name);
    } else {
      res = CheckFutureArrivals(name, date);
    }
    if (res != null) {
      ConfirmationNumber cn = res.Confirmation;
      return cn;
    } else {
      if (createIfNeeded) {
        Reservation newRes = new Reservation();
        //... build reservation ...
        ConfirmationNumber cn = newRes.Confirmation;
        return cn;
      } else {
        return null;
      }
<return type>     }
<return type>   }
  //...etc...
}///:~
```

The first step in **Reservation.ConfirmReservation( )** is to see if the passed-in **Date** is today; the result of this check activates different control paths, **CheckTodaysArrivals( )** or **CheckFutureArrivals( )**. While the behavior of **Reservation.ConfirmReservation( )** changes depending on the value of

Date, this is *not* logical cohesion because the test and the control is within **Reservation.ConfirmReservation( )**.

However, the **bool createIfNeeded** argument is ugly. It makes unit-testing **Reservation.ConfirmReservation( )** literally twice as hard.

Logical association can almost always be transformed into at least procedural association. This first example shows code that might call **Reservation.ConfirmReservation( )** as it's written:

```
class TravelAgent{
  void ConfirmPackage(Person customer){
    bool create = customer.CreateResIfNull();
    foreach(DateTime d in customer.TravelDates){
      ConfirmationNumber cn =
        Reservation.ConfirmReservation
        (customer.Name, d, create);
      if (cn != null) {
        customer.AddConfirmation(cn);
      }
    }
  }
}
```

But a first step towards improving the code would be to refactor from logical to procedural association:

```
//:c09:RefactoredReservations.cs
using System;

class Reservation {
  internal Reservation(string name, DateTime d){
    //...etc...
  }
  public static ConfirmationNumber ConfirmReservation
  (string name, DateTime date){
    //...etc...
    return cn;
  }

  ConfirmationNumber cn;
  public ConfirmationNumber Confirmation{
    get { return cn; }
```

```
    set { cn = value; }
  }
}

class TravelAgent {
  public void ConfirmPackage(Person customer){
    bool create = customer.CreateResIfNull();
    foreach(DateTime d in customer.TravelDates){
      string name = customer.Name;
      ConfirmationNumber cn =
        Reservation.ConfirmReservation(name, d);
      if (cn == null && create == true) {
        Reservation res =
          new Reservation(name, d);
        cn = res.Confirmation;
      }
      if (cn != null) {
        customer.AddConfirmation(cn);
      }
    }
  }
}
```

While this new version of **TravelAgent.ConfirmPackage( )** may still not be ideal, where **bool create** is declared, assigned, and how it is used to affect behavior is all localized within **TravelAgent.ConfirmPackage( )**. You can see exactly where it comes from and what it affects. With the original, logically associated versions of these methods, this was far less apparent.

Coincidental cohesion

Coincidental association isn't very common; it occurs when a method does two totally unrelated tasks:

```
void PayTaxesAndPickUpALoafOfBread(){
  PayTaxes();
  PickUpALoafOfBread();
}
```

The one place where coincidental cohesion is seen fairly commonly is with programmers who over-use graphical forms as a stand-in for object-oriented design. This will be discussed in great detail in Chapter 14's discussion of user-interface architectures.

Design is as design does

The end-user has no interest in your software's design. They care that your software helps them do their job easier. They care that your software doesn't irritate them. They care that when they talk to you about what they need from the software, you listen and respond sympathetically (and in English, not Geek). They care that your software is cheap and reliable and robust. Other than that, they don't care if it's written in assembly language, LISP, C#, or FORTRAN, and they sure as shooting don't care about your class and sequence diagrams.

So there's really *nothing* that matters about design except how easy it is to change or extend – the things that you have to do when you discover that your existing code falls short in some manner.

So how do you design for change?

First, do no harm

It's acceptable for you to decide not to make a change. It's acceptable for your change to turn out to be less helpful to the user than the original. It's acceptable for your change to be dead wrong, either because of miscommunication or because your fingers slipped on the keys and typed a plus instead of a minus. What is unacceptable is for you to commit a change that breaks something else in your code. And it is unacceptable for you to not know if your change breaks something else in the code. Yet such a pathological state, where the only thing that developers can say about their code is a pathetic "Well, it *shouldn't* cause anything else to change," is very common. Almost as common are situations where changes do, in fact, seem to randomly affect the behavior of the system as a whole.

It is counterintuitive, but the number one thing you can do to speed up your development schedule is to write a lot of tests. It doesn't pay off in the first days, but it pays off in just a few weeks. By the time you get several months into a project, you will be madly in love with your testing suite. Once you develop a system with a proper suite of tests, you will consider it incompetent foolishness to develop without one.

With modern languages that support reflection, it has become possible to write test infrastructure systems that discover at runtime what methods are tests (by convention, methods that begin with the string "test") and running them and reporting the results. The predominant such system is JUnit, created by Kent Beck and Erich Gamma. Several ports and refactorings of JUnit for .NET

languages are available, the most popular of which is Philip Craig's NUnit. Appendix C in this book goes into some detail on the use of NUnit.

Software systems are among the most complex structures built by humans, and unintended consequences are an inevitable part of manipulating those systems. The speed with which you can change a software system is dependent on many things, but on nothing so much as your ability to isolate a change in both time and program structure. Extensive unit tests, runnable by a batch process, is by far the best way to do this.

Write boring code

A good application is interesting; good code is boring. The most boring code has no cyclomatic complexity, no coupling, and functional cohesion. What can you say about:

```
///<summary>Adds two integers</summary>
///<exception cref="System.OverflowException">
///Thrown if x+y > Int32.MaxValue or < Int32.MinValue
///</exception>
public int Add(int x, int y){
  return x + y;
}
```

that goes beyond the code? Even if this were a core method called a million times per second, you could entrust to a junior programmer the task of modifying it to work with 64-bit **long** values.

At the other extreme is "clever code." Clever code is characterized by high cyclomatic complexity and extensive coupling. Often, clever code is written in an ill-advised attempt to increase performance. Often, clever code runs slower than boring code.

Make names meaningful

Type, method, and variable names should all be as descriptive as possible. This is an area where the book's examples are not a good guide; the 54 character columns of the book dictate that fully descriptive, fully spelled-out names can't be used.

Needless to say, names must also be accurate. This is an obvious quality feature and yet people skimp on it, as it requires a search-and-replace that may have to span multiple files. Such an operation is no big deal with any kind of decent programming editor, but occasionally "clever code" interferes. For instance, one

of us (Larry) was once stymied to see a testing suite break after a class was renamed during refactoring; it turned out that during initialization, the names of classes that implemented a filtering interface were read from a configuration file and dynamic class loading used to instantiate the desired filters; the configuration file contained the old name. This is another lesson in the importance of unit-testing: usually one expects that a clean compile is all that is necessary to confirm a renaming operation, but a testing suite will flush out unexpected problems such as this in a matter of minutes, not the hours or days that might be required if the "obviously working" change had been checked in to source-code control.

Classes should be named with the noun of the domain concept that is being encapsulated: **Car**, **Profit**, or **EducationalObjective**. You should not repeat the base-class name in descendant classes: **Compact** and **FourDoor** are good names for classes descending from **Car**, not **CompactCar** and **FourDoorCar**. An alternative can be used sparingly: sometimes a class does not correspond to a domain concept but is created strictly to play a role in a design pattern. In this situation, the class may be named according to the design role: **ConcreteFlyweight** or **CompositeElement** are acceptable names for classes that are playing particular roles in the *Flyweight* and *Composite* design patterns. Better, even in this situation you should try to tie the role and the domain together: perhaps **ConcreteGlyphFlyweight** or **CompositeGraphicElement**.

Methods that return **void** should be named with a strong verb: **Document.Print( )**. If a method returns a value, the method name should be a noun-based description of the return value: **Invoice.SalesTax( )**. If a method is difficult to describe with strong words, there's probably a problem with its cohesion. If you can't come up with something better than **Execute( )** or **DoCalculation( )**, or if the method name includes conjunctions (**Invoice.CalcFinalCostAndSendShippingOrder( )**), it almost certainly has procedural cohesion or worse.

Sometimes, you'll find that your class is like some standard library class. Even if after consideration you decide that your class should not descend from the library class, you should strongly consider naming (and implementing!) its methods to correspond to those in the library class.

Limit complexity

The number of lines in a method is not a good indicator that you should split it into two, but the cyclomatic complexity of the method is. If you use Visual Studio

.NET to program Windows Forms, it will place all the code relating to constructing the user-interface into a method called **InitializeComponent( )**; this method may be hundreds of lines long, but it contains no control-flow operators, so it's length is irrelevant. On the other hand, the 15 lines of this leap year calculation are about as complex as is acceptable:

```
//:c09:LeapYearCalc.cs
using System;

class LeapYearCalc {
  static bool LeapYear(int year){
    if (year % 4 != 0) {
      return false;
    } else {
      if (year % 400 == 0) {
        return true;
      } else {
        if (year % 100 == 0) {
          return false;
        } else {
          return true;
        }
      }
    }
  }

  public static void Test(int year, bool val){
    if (val == LeapYear(year)) {
      Console.WriteLine(
        "{0} correctly calced as {1}", year, val);
       return;
    }
    throw new TestFailedException(
      String.Format("{0} not calc'ed as {1}", year, val) );
  }

  public static void Main(){
    Test(1999, false);
    Test(2000, true);
    Test(1900, false);
  }
```

```
}

class TestFailedException : ApplicationException{
  public TestFailedException(String s): base(s){ }
}///:~
```

Some simple testing code is shown because, less than a month before this book went to press, we found a bug in the **LeapYearCalc( )** function had! So maybe the 15 lines in that function are a little *more* complex than allowable…

Make stuff as private as possible

Now that we've introduced the concept of coupling and cohesion, the use of the visibility modifiers in C# should be more compelling. The more visible a piece of data, the more available it is to be used for common coupling or communicational and worse forms of cohesion.

The very real advantages that come from object-orientation, C#, and the .NET Framework do not derive from the **noun.Verb( )** form of method calls or from using brackets to specify scope. The success of the object-oriented paradigm stems from *encapsulation*, the logical organization of data and behavior *with restricted access*. Coupling and cohesion are more precise terms to discuss the benefits of encapsulation, but class interfaces, inheritance, the visibility modifiers, and Properties – the purpose of all of these things is to hide a large number of implementation details while simultaneously providing functionality and extensibility.

Why do details need to be hidden? For the original programmer, details that are out of sight are out of mind, and the programmer frees some amount of his or her finite mental resources for work on the *next* issue. More importantly than this, though, details need to be hidden so software can be tested, modified, and extended. Programming is a task that is characterized by continuously overcoming failure: a missed semicolon at the end of a line, a typo in a name, a method that fails a unit test, a clumsy design, a customer who says "this isn't what I wanted." So as a programmer you are *always* revisiting existing work, whether it's three minutes, three weeks, or three years old. Your productivity as a professional programmer is not governed by how fast you can create, it is governed by how fast you can fix. And the speed with which you can fix things is influenced by the number of details that must be characterized as relevant or irrelevant. Objects localize and isolate details.

Coupling, cohesion, and design trends

Coupling and cohesion, popularized by Ed Yourdon and Larry Constantine way back in the 1970s, are still the best touchstones for determining whether a method or type is built well or poorly. The most important software engineering book of the 1990s was *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the "Gang of Four"). What really set *Design Patterns* apart is that it was based on an archaeological approach to design; instead of putting their no-doubt-clever heads together and saying "Here's a new way to solve this problem," the book documents common structures and interactions (design patterns) that they *found* in proven software systems. When compared to other object-oriented design books, what leaps out about *Design Patterns* is the complete lack of references to objects that correspond to physical items in the real world and the recurring emphasis of techniques to decrease coupling and increase cohesion.

An interesting question is whether low coupling and high cohesion are a *cause* of good design or a *consequence* of it. The traditional view has been that they are a consequence of design: you go into your cubicle, fire up your CASE tool, think deep thoughts, and emerge with a set of diagrams that will wow the crowds at the design review. This view is challenged by one of the better books of the past few years: Martin Fowler's *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999). This book makes the fairly radical claim that taking "simple, even simplistic" steps on existing code, no matter how chaotic, *leads to* good design. Fowler goes even further and points out that without refactoring, the design of a system decays over time as the system is maintained; this is one of those obvious-in-retrospect observations that invalidates an entire worldview, in this case, the worldview that design is done with a diagramming tool and a blank piece of paper.

Refactoring is changing the internal structure of your code without changing its internal behavior; Fowler presents a suite of refactorings and "code smells" to indicate when refactoring is needed. The book doesn't explicitly address issues of

coupling and cohesion[5], but when viewed through the lens of structured design, refactoring is clearly driven by these concerns.

Summary

Any software project of more than a few hundred lines of code should be organized by a principle. This principle is called the software's *architecture*. The word architecture is used in many ways in computing; software architecture is a characteristic of code structure and data flows between those structures. There are many proven software architectures; object-orientation was originally developed to aid in simulation architectures but the benefits of objects are by no means limited to simulations.

Many modern-day projects are complex enough that it is appropriate to distinguish between the architecture of the overall systems and the architecture of different subsystems. The most prevalent examples of this are Web-based systems with rich clients, where the system as a whole is often an *n*-tier architecture, but each tier is a significant project in itself with its own organizing principle.

Where the aims of architecture are strategic and organizational, the aims of software design are tactical and pragmatic. The purpose of software design is to iteratively deliver client value as inexpensively as possible. The most important word in that previous sentence is "iteratively." You may fool yourself into believing that design, tests, and refactoring are wastes of time on the current iteration, but you can't pretend that they are a waste of time if you accept that whatever you're working on is likely to be revisited every three months, especially if you realize that if you don't make things clear, they're going to be going to be calling *you* at 3 o'clock in the morning when the Hong Kong office says the system has frozen[6].

Software design decisions, which run the gamut from the parameters of a method to the structure of a namespace, are best made by consideration of the principles of coupling and cohesion. Coupling is the degree to which two software elements are interdependent; cohesion is a reflection of a software element's internal

---

[5] Like *Extreme Programming*, another excellent recent book, *Refactoring* promotes homespun phrases like "code smells" and "the rule of three" that are no more or less exclusionary than the software engineering jargon they pointedly avoid.

[6] Actually, they'll call the IT guys first. That's why it's important to cultivate the perception that you know absolutely nothing about system administration and hardware.

dependencies. Good software designs are characterized by loose coupling and high cohesion. With the rise of object orientation, the word “encapsulation” has come to be used to characterize all of the benefits of detail hiding, high cohesion, and loose coupling.

At this halfway point in the book, we have covered C# as a language and the concepts of object-orientation. However, we’ve hardly scratched the surface of the .NET Framework SDK, hundreds of classes and namespaces that provide an object-oriented view of everything from data structures to user-interfaces to the World Wide Web. From hereon out, the concerns of the book are generally less specific to the C# language per se and more generally applicable to the capabilities that the .NET Framework would make available to any language. This does not mean that we’ve exhausted our discussion of the C# language, however. Some of the most interesting aspects of the C# language are yet to be introduced.

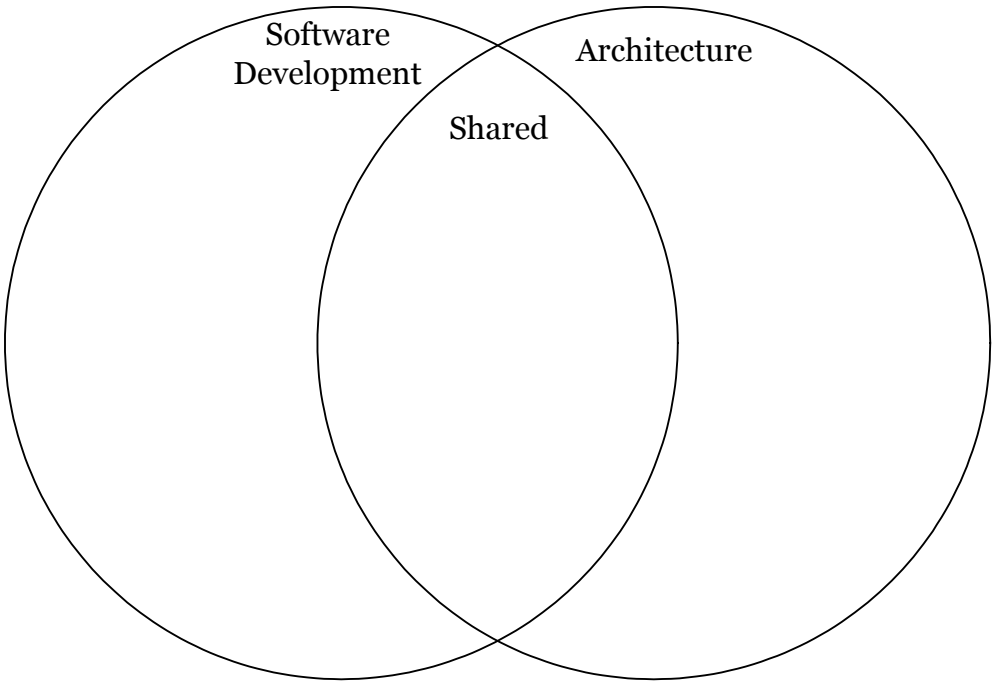
Exercises

1. Try pair programming on one of the problems in the party domain. Try to reserve judgment until you've paired with programmers who are more, less, and similarly experienced.

2. Read Appendix C, “Test-First Programming with NUnit” and tackle a simple task in the party domain via test-first programming.

3. Write a one-page essay evaluating your personal experience with pair and test-first programming.

4. Fill in the following Venn diagram comparing aspects of software development with physical architecture.



5. Write a one-page essay defending or refuting the statement “Software is architecture.”

6. The hardware manufacturers are thrilled with your work with the robotic party servant and want you to lead the development of all the robot's behavioral software. What kind of architecture will you adopt? Why?

7. Evaluate your party servant system. Use everything that you have learned to improve your design and implementation.

10: Collecting Your Objects

It's a fairly simple program that has only a fixed quantity of objects with known lifetimes.

In general, your programs will always be creating new objects based on some criteria that will be known only at the time the program is running. You won't know until run-time the quantity or even the exact type of the objects you need. To solve the general programming problem, you need to be able to create any number of objects, anytime, anywhere. So you can't rely on creating a named reference to hold each one of your objects:

```
MyType myObject;
```

since you'll never know how many of these you'll actually need.

To solve this rather essential problem, C# has several ways to hold objects (or rather, references to objects). The built-in type is the array, which has been discussed before. Also, the C# System.Collections namespace has a reasonably complete set of *container classes* (also known as *collection classes*). Containers provide sophisticated ways to hold and manipulate your objects.

Containers open the door to the world of computing with data structures, where amazing results can be achieved by manipulating the abstract geometry of trees, vector spaces, and hyperplanes. While data structure programming lies outside of the workaday world of most programmers, it is very important in scientific, graphic, and game programming.

Arrays

Most of the necessary introduction to arrays was covered in Chapter 5, which showed how you define and initialize an array. Holding objects is the focus of this chapter, and an array is just one way to hold objects. But there is a number of other ways to hold objects, so what makes an array special?

There are two issues that distinguish arrays from other types of containers: efficiency and type. The array is the most efficient way that C# provides to store

and randomly access a sequence of objects (actually, object references). The array is a simple linear sequence, which makes element access fast, but you pay for this speed: when you create an array object, its size is fixed and cannot be changed for the lifetime of that array object. You might suggest creating an array of a particular size and then, if you run out of space, creating a new one and moving all the references from the old one to the new one. This is the behavior of the **ArrayList** class, which will be studied later in this chapter. However, because of the overhead of this size flexibility, an **ArrayList** is measurably less efficient than an array.

The **vector** container class in C++ *does* know the type of objects it holds, but it has a different drawback when compared with arrays in C#: the C++ **vector**'s **operator[]** doesn't do bounds checking, so you can run past the end[1]. In C#, you get bounds checking regardless of whether you're using an array or a container—you'll get an **IndexOutOfRangeException** if you exceed the bounds. As you'll learn in Chapter 11, this type of exception indicates a programmer error, and thus you don't need to check for it in your code. As an aside, the reason the C++ **vector** doesn't check bounds with every access is speed—in C# you have the performance overhead of bounds checking all the time for both arrays and containers.

The other generic container classes that will be studied in this chapter, **ICollection, IList** and **IDictionary**, all deal with objects as if they had no specific type. That is, they treat them as type **object**, the root class of all classes in C#. This works fine from one standpoint: you need to build only one container, and any C# object will go into that container. This is the second place where an array is superior to the generic containers: when you create an array, you create it to hold a specific type. This means that you get compile-time type checking to prevent you from putting the wrong type in, or mistaking the type that you're extracting. Of course, C# will prevent you from sending an inappropriate message to an object, either at compile-time or at run-time. So it's not much riskier one way or the other; it's just nicer if the compiler points it out to you, faster at run-time, and there's less likelihood that the end user will get surprised by an exception.

Typed generic classes (sometimes called "parameterized types" and sometimes just "generics") are not part of the initial .NET framework but will be. Unlike C++'s templates or Java's proposed extensions, Microsoft wishes to implement support for "parametric polymorphism" within the Common Language Runtime itself. Don Syme and Andrew Kennedy of Microsoft's Cambridge (England) Research Lab

---

[1] It's possible, however, to ask how big the **vector** is, and the **at( )** method *does* perform bounds checking.

published papers in Spring 2001 on a proposed strategy and Anders Hjelsberg hinted at C#'s Spring 2002 launch that implementation was well under way.

For the moment, though, efficiency and type checking suggest using an array if you can. However, when you're trying to solve a more general problem arrays can be too restrictive. After looking at arrays, the rest of this chapter will be devoted to the container classes provided by C#.

Arrays are first-class objects

Regardless of what type of array you're working with, the array identifier is actually a reference to a true object that's created on the heap. This is the object that holds the references to the other objects, and it can be created either implicitly, as part of the array initialization syntax, or explicitly with a **new** expression. Part of the array object is the read-only **Length** property that tells you how many elements can be stored in that array object. For rectangular arrays, the **Length** property tells you the total size of the array, the **Rank** property tells you the number of dimensions in the array, and the **GetLength(int)** method will tell you how many elements are in the given rank.

The following example shows the various ways that an array can be initialized, and how the array references can be assigned to different array objects. It also shows that arrays of objects and arrays of primitives are almost identical in their use. The only difference is that arrays of objects hold references, while arrays of primitives hold the primitive values directly.

```
//:c10:ArraySize.cs
// Initialization & re-assignment of arrays.
using System;

class Weeble {
} // A small mythical creature

public class ArraySize {
  public static void Main() {
    // Arrays of objects:
    Weeble[] a; // Null reference
    Weeble[] b = new Weeble[5]; // Null references
    Weeble[,] c = new Weeble[2, 3]; //Rectangular array
    Weeble[] d = new Weeble[4];
    for (int index = 0; index < d.Length; index++)
      d[index] = new Weeble();
    // Aggregate initialization:
```

```
Weeble[] e = {
  new Weeble(), new Weeble(), new Weeble()
};
// Dynamic aggregate initialization:
a = new Weeble[] {
  new Weeble(), new Weeble()
};
// Square dynamic aggregate initialization:
c = new Weeble[,] {
  { new Weeble(), new Weeble(), new Weeble()},
  { new Weeble(), new Weeble(), new Weeble()}
};

Console.WriteLine("a.Length=" + a.Length);
Console.WriteLine("b.Length = " + b.Length);
Console.WriteLine("c.Length = " + c.Length);
for (int rank = 0; rank < c.Rank; rank++) {
  Console.WriteLine(
    "c.Length[{0}] = {1}", rank, c.GetLength(rank));
}
// The references inside the array are
// automatically initialized to null:
for (int index = 0; index < b.Length; index++)
  Console.WriteLine("b[" + index + "]=" + b[index]);
Console.WriteLine("d.Length = " + d.Length);
Console.WriteLine("d.Length = " + d.Length);
a = d;
Console.WriteLine("a.Length = " + a.Length);

// Arrays of primitives:
int[] f; // Null reference
int[] g = new int[5];
int[] h = new int[4];
for (int index = 0; index < h.Length; index++)
  h[index] = index*index;
int[] i = { 11, 47, 93};
// Compile error: Use of unassigned local variable 'f'
//! Console.WriteLine("f.Length=" + f.Length);
Console.WriteLine("g.Length = " + g.Length);
// The primitives inside the array are
// automatically initialized to zero:
```

```
    for (int index = 0; index < g.Length; index++)
      Console.WriteLine("g[" + index + "]=" + g[index]);
    Console.WriteLine("h.Length = " + h.Length);
    Console.WriteLine("i.Length = " + i.Length);
    f = i;
    Console.WriteLine("f.Length = " + f.Length);
    f = new int[] { 1, 2};
    Console.WriteLine("f.Length = " + f.Length);
  }
} ///:~
```

Here's the output from the program:

```
a.Length=2
b.Length = 5
c.Length = 6
c.Length[0] = 2
c.Length[1] = 3
b[0]=
b[1]=
b[2]=
b[3]=
b[4]=
d.Length = 4
d.Length = 4
a.Length = 4
g.Length = 5
g[0]=0
g[1]=0
g[2]=0
g[3]=0
g[4]=0
h.Length = 4
i.Length = 3
f.Length = 3
f.Length = 2
```

The array **a** is initially just a **null** reference, and the compiler prevents you from doing anything with this reference until you've properly initialized it. The array **b** is initialized to point to an array of **Weeble** references, but no actual **Weeble** objects are ever placed in that array. However, you can still ask what the size of the array is, since **b** is pointing to a legitimate object. This brings up a slight drawback: you can't find out how many elements are actually *in* the array, since **Length** tells you only

how many elements *can* be placed in the array; that is, the size of the array object, not the number of elements it actually holds. However, when an array object is created its references are automatically initialized to **null**, so you can see whether a particular array slot has an object in it by checking to see whether it's **null**. Similarly, an array of primitives is automatically initialized to zero for numeric types, **(char)0** for **char**, and **false** for **bool**.

Array **c** shows the creation of the array object followed by the assignment of **Weeble** objects to all the slots in the array. Array **d** shows the "aggregate initialization" syntax that causes the array object to be created (implicitly with **new** on the heap, just like for array **c**) *and* initialized with **Weeble** objects, all in one statement.

The next array initialization could be thought of as a "dynamic aggregate initialization." The aggregate initialization used by **d** must be used at the point of **d**'s definition, but with the second syntax you can create and initialize an array object anywhere. For example, suppose **Hide( )** is a method that takes an array of **Weeble** objects. You could call it by saying:

```
Hide(d);
```

but you can also dynamically create the array you want to pass as the argument:

```
Hide(new Weeble[] { new Weeble(), new Weeble() });
```

In some situations this new syntax provides a more convenient way to write code.

Rectangular arrays are initialized using nested arrays. Although a rectangular array is contiguous in memory, C#'s compiler will not allow you to ignore the dimensions; you cannot cast a flat array into a rectangular array or initialize a rectangular array in a "flat" manner.

The expression:

```
a = d;
```

shows how you can take a reference that's attached to one array object and assign it to another array object, just as you can do with any other type of object reference. Now both **a** and **d** are pointing to the same array object on the heap.

The second part of **ArraySize.cs** shows that primitive arrays work just like object arrays *except* that primitive arrays hold the primitive values directly.

The Array class

In **System.Collections**, you'll find the **Array** class, which has a variety of interesting properties and methods. **Array** is defined as implementing **ICloneable, IList, ICollection,** and **IEnumerable**. This is actually a pretty sloppy declaration, as **IList** is declared as extending **ICollection** and **IEnumerable**, while **ICollection** is itself declared as extending **IEnumerable** (Figure 10-1)!

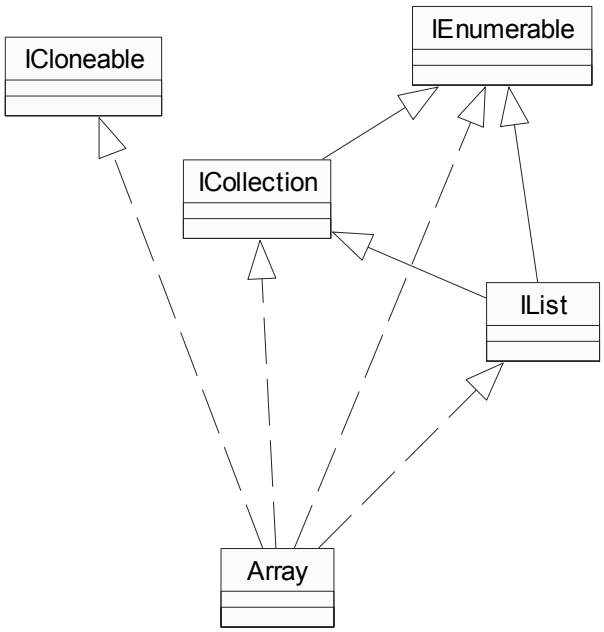


Figure 10-1: The Array class has a complex set of base types

The **Array** class has some properties inherited from **ICollection** that are the same for all instances: **IsFixedSize** is always **true**, **IsReadOnly** and **IsSynchronized** are always **false.**

Array's static methods

The **Array** class has several useful static methods, which are illustrated in this program:

```
//:c10:ArrayStatics.cs
using System;
using System.Collections;
```

```
class Weeble {
  string name;
  internal string Name{
    get { return name;}
    set { name = value;}
  }
  internal Weeble(string name) {
    this.Name = name;
  }
}
class ArrayStatics {
  static string[] dayList = new string[]{
    "sunday", "monday", "tuesday", "wednesday",
    "thursday", "friday", "saturday"
  };

  static string[,] famousCouples = new string[,]{
    { "George", "Martha"}, { "Napolean", "Josephine"},
    { "Westley","Buttercup"}
  };

  static Weeble[] weebleList = new Weeble[]{
    new Weeble("Pilot"), new Weeble("Firefighter")
  };

  public static void Main() {
    //Copying arrays
    Weeble[] newList = new Weeble[weebleList.Length];
    Array.Copy(weebleList, newList, weebleList.Length);
    newList[0] = new Weeble("Nurse");
    bool newReferences = newList[0] != weebleList[0];
    Console.WriteLine("New references ==  "
      + newReferences);
    //Copying a rectangular array works
    string[,] newSquareArray =
    new string[famousCouples.GetLength(0),
      famousCouples.GetLength(1)];
    Array.Copy(famousCouples, newSquareArray,
      famousCouples.Length);
```

```
    //In-place sorting
    string[] sortedDays = new string[dayList.Length];
    Array.Copy(dayList, sortedDays, dayList.Length);
    Array.Sort(sortedDays);
    for (int i = 0; i < sortedDays.Length; i++) {
      Console.WriteLine("sortedDays[{0}] = {1}",
        i, sortedDays[i]);
    }
    //Binary search of sorted 1-D Array
    int tuesdayIndex =
    Array.BinarySearch(sortedDays, "tuesday");
    Console.WriteLine(
      "dayList[{0}] == \"tuesday\"", tuesdayIndex);
    //! int georgeIndex =
    //! Array.BinarySearch(famousCouples, "George"};
    // Causes compile error

    //Reverse
    Array.Reverse(sortedDays);
    for (int i = 0; i < sortedDays.Length; i++) {
      Console.WriteLine(
        "Reversed sortedDays[{0}] = {1}",
        i, sortedDays[i]);
    }

    //Quickly erasing an array section,
    //even if multidimensional
    Array.Clear(famousCouples, 2, 3);
    for(int x = 0; x < famousCouples.GetLength(0); x++)
      for(int y = 0; y < famousCouples.GetLength(1);
        y++)
        Console.WriteLine(
          "FamousCouples[{0},{1}] = {2}",
          x, y, famousCouples[x,y]);
  }
}///:~
```

After declaring a Weeble class (this time with a Name property to make them easier to distinguish), the **ArrayStatics** class declares several static arrays – **dayList** and **weebleList**, which are both one-dimensional, and the square **famousCouples** array.

Array.Copy( ) provides a fast way to copy an array (or a portion of it). The new array contains all new references, so changing a value in your new list will not change the value in your original, as would be the case if you did:

```
Weeble[] newList = weebleList;
newList[0] = new Weeble("Nurse");
```

Array.Copy( ) works with multidimensional arrays, too. The program uses the **GetLength(int)** method to allocate sufficient storage for the new **SquareArray**, but then uses the **famousCouples.Length** property to specify the size of the copy. Although **Copy( )** seems to "flatten" multidimensional arrays, using arrays of different rank will throw a runtime RankException.

The static method **Array.Sort( )** does an in-place sort of the array's contents and **BinarySearch( )** provides an efficient search on a sorted array.

Array.Reverse( ) is self-explanatory, but **Array.Clear( )** has the perhaps surprising behavior of slicing across multidimensional arrays. In the program, **Array.Clear(famousCouples, 2, 3)** treats the multidimensional **famousCouples** array as a flat array, setting to **null** the values of indices [1,0], [1,1], and [2,0].

Array element comparisons

How does **Array.Sort( )** work? A problem with writing generic sorting code is that sorting must perform comparisons based on the actual type of the object. Of course, one approach is to write a different sorting method for every different type, but you should be able to recognize that this does not produce code that is easily reused for new types.

A primary goal of programming design is to "separate things that change from things that stay the same," and here, the code that stays the same is the general sort algorithm, but the thing that changes from one use to the next is the way objects are compared. So instead of hard-wiring the comparison code into many different sort routines, the *Strategy Pattern* is used. In the Strategy Pattern, the part of the code that varies from case to case is encapsulated inside its own class, and the part of the code that's always the same makes a call to the part of the code that changes. That way you can make different objects to express different strategies of comparison and feed them to the same sorting code.

In C#, comparisons are done by calling back to the **CompareTo( )** method of the **IComparable** interface. This method takes another **object** as an argument, and produces a negative value if the current object is less than the argument, zero if the

argument is equal, and a positive value if the current object is greater than the argument.

Here's a class that implements **IComparable** and demonstrates the comparability by using **Array.Sort( )**:

```
//:c10:CompType.cs
// Implementing IComparable in a class.
using System;

public class CompType: IComparable {
  int i;
  int j;
  public CompType(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public override string ToString() {
    return "[i = " + i + ", j = " + j + "]";
  }

  public int CompareTo(Object rv) {
    int rvi = ((CompType)rv).i;
    if (i > rvi)
      return 1;
    else if (i == rvi)
      return 0;
    else
      return -1;
(i < rvi ? -1 : (i == rvi ? 0 : 1));
  }

  private static Random r = new Random();

  private static void ArrayPrint(String s, Array a){
    Console.Write(s);
    foreach(Object o in a){
      Console.Write(o + ",");
    }
    Console.WriteLine();
  }
```

```
  public static void Main() {
    CompType[] a = new CompType[10];
    for (int i = 0; i < 10; i++) {
      a[i] = new CompType(r.Next(100), r.Next(100));
    }
    ArrayPrint("Before sorting, a = ", a);
    Array.Sort(a);
    ArrayPrint("After sorting, a = ", a);
  }
} ///:~
```

When you define the comparison function, you are responsible for deciding what it means to compare one of your objects to another. Here, only the **i** values are used in the comparison, and the **j** values are ignored.

The **Main( )** method creates a bunch of **CompType** objects that are initialized with random values and then sorted. If **Comparable** hadn't been implemented, then you'd get an InvalidOperationException thrown at runtime when you tried to call **Array.Sort( )**.

What? No bubbles?

In the not-so-distant past, the sort and search methods used in a program were a matter of constant debate and anguish. In the good old days, even the most trivial datasets had a good chance of being larger than RAM (or "core" as we used to say) and required intermediate reads and writes to storage devices that could take, yes, seconds to access (or, if the tapes needed to be swapped, minutes). So there was an enormous amount of energy put into worrying about internal (in-memory) versus external sorts, the stability of sorts, the importance of maintaining the input tape until the output tape was verified, the "operator dismount time," and so forth.

Nowadays, 99% of the time you can ignore the particulars of sorting and searching. In order to get a decent idea of sorting speed, this program requires an array of 1,000,000 elements, and still it executes in a matter of seconds:

```
//:c10:FastSort.cs
using System;

class Sortable : IComparable {
  int i;
  internal Sortable(int i) {
    this.i = i;
  }
```

```
  public int CompareTo(Object o) {
    try {
      Sortable s = (Sortable) o;
      return i = s.i;
    } catch (InvalidCastException) {
      throw new ArgumentException();
    }
  }
}

class SortingTester {
  static TimeSpan TimedSort(IComparable[] s){
    DateTime start = DateTime.Now;
    Array.Sort(s);
    TimeSpan duration = DateTime.Now - start;
    return duration;
  }
  public static void Main() {
    for (int times = 0; times < 10; times++) {
      Sortable[] s = new Sortable[1000000];
      for (int i = 0; i < s.Length; i++) {
        s[i] = new Sortable(i);
      }
      Console.WriteLine("Time to sort already sorted"
        + " array: " + TimedSort(s));
      Random rand = new Random();
      for (int i = 0; i < s.Length; i++) {
        s[i] = new Sortable(rand.Next());
      }
      Console.WriteLine("Time to sort mixed up array: "
        + TimedSort(s));
    }
  }
}///:~
```

The results show that **Sort( )** works faster on an already sorted array, which indicates that behind the scenes, it's probably using a merge sort instead of QuickSort. But the sorting algorithm is certainly less important than the fact that a computer that costs less than a thousand dollars can perform an in-memory sort of a million-item array! Moore's Law has made anachronistic an entire field of

knowledge and debate that seemed, not that long ago, fundamental to computer programming.

This is an important lesson for those who wish to have long careers in programming: never confuse the mastery of today's facts with preparation for tomorrow's changes. Within a decade, we will have multi-terabyte storage on the desktop, trivial access to distributed teraflop processing, and probably specialized access to quantum computers of significant capability. Eventually, although probably not within a decade, there will be breakthroughs in user interfaces and we'll abandon the keyboard and the monitor for voice and gesture input and "augmented reality" glasses. Almost all the programming facts that hold today will be as useless as the knowledge of how to do an oscillating sort with criss-cross distribution. A programmer must never stand still.

Unsafe arrays

Despite the preceding discussion of the steady march of technical obsolescence, the facts on the ground often agitate towards throwing away the benefits of safety and abstraction and getting closer to the hardware in order to boost performance. Often, the correct solution in this case will be to move out of C# altogether and into C++, a language which will continue for some time to be the best for the creation of device drivers and other close-to-the-metal components.

However, manipulating arrays can sometimes introduce bottlenecks in higher-level applications, such as multimedia applications. In such situations, unsafe code may be worthwhile. The basic impetus for using unsafe arrays is that you wish to manipulate the array as a contiguous block of memory, foregoing bounds checking.

As a testbed for exploring performance with unsafe arrays, we'll use a transformation that actually has tremendous practical applications. Wavelet transforms are fascinating and their utility has hardly been scratched. The simplest transform is probably the two-dimensional Haar transform on a matrix of doubles. The Haar transform converts a list of values into the list's average and differences, so the list {2, 4} is transformed into {3, 1} == {(2 + 4) / 2, ((2 + 4) / 2) – 2}. A two-dimensional transform just transforms the rows and then the columns, so {{2, 4},{5,6}} becomes {{4.25, .75},{1.25, -0.25}}:

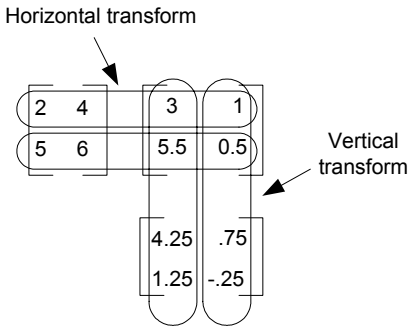


Figure 10-2: The Haar transform is a horizontal followed by vertical transform

Wavelets have many interesting characteristics, including being the basis for some excellent compression routines, but are expensive to compute for arrays that are typical of multimedia applications, especially because to be useful they are usually computed $\log_2$(MIN(dimension size)) times per array!

The following program does such a transform in two different ways, one a safe method that uses typical C# code and the other using unsafe code.

```
//:c10:FastBitmapper1.cs
using System;
using System.IO;

namespace FastBitmapper{
  public interface Transform{
    void HorizontalTransform(double[,] matrix);
    void VerticalTransform(double[,] matrix);
  }

  public class Wavelet {
    public void Transform2D(double[,] matrix,
      Transform t) {
      int minDimension = matrix.GetLength(0);
      if (matrix.GetLength(1) < minDimension)
        minDimension = matrix.GetLength(1);
      int levels =
        (int) Math.Floor(Math.Log(minDimension, 2));
      Transform2D(matrix, levels, t);
    }

    public void Transform2D(double[,] matrix,
```

```
      int steps, Transform tStrategy) {
      for (int i = 0; i < steps; i++) {
        tStrategy.HorizontalTransform(matrix);
        tStrategy.VerticalTransform(matrix);
      }
    }

    public void TestSpeed(Transform t) {
      Random rand = new Random();
      double[,] matrix = new double[2000,2000];
      for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++) {
          matrix[i,j] = rand.NextDouble();
        }
      DateTime start = DateTime.Now;
      this.Transform2D(matrix, t);
      TimeSpan dur = DateTime.Now - start;
      Console.WriteLine(
        "Transformation with {0} took {1} ",
        t.GetType().Name, dur);
    }

    public static void Main() {
      Wavelet w = new Wavelet();
      for (int i = 0; i < 10; i++) {
        //Get things right first
        w.TestSpeed(new SafeTransform());
        //Have not defined UnsafeTransform yet
        //! w.TestSpeed(new UnsafeTransform());
      }
    }
  }

  internal class SafeTransform : Transform {
    private void Transform(double[] array) {
      int halfLength = array.Length >> 1;
      double[] avg = new double[halfLength];
      double[] diff = new double[halfLength];
      for (int pair = 0; pair < halfLength; pair++) {
        double first = array[pair * 2];
        double next = array[pair * 2 + 1];
```

```
        avg[pair] = (first + next) / 2;
        diff[pair] = avg[pair] - first;
      }
      for (int pair = 0; pair < halfLength; pair++) {
        array[pair] = avg[pair];
        array[pair + halfLength] = diff[pair];
      }
    }

    public void HorizontalTransform(double[,] matrix) {
      int height = matrix.GetLength(0);
      int width = matrix.GetLength(1);
      double[] row = new double[width];
      for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
          row[j] = matrix[i, j];
        }
        Transform(row);
        for (int j = 0; j < width; j++) {
          matrix[i,j] = row[j];
        }
      }
    }

    public void VerticalTransform(double[,] matrix) {
      int height = matrix.GetLength(0);
      int length = matrix.GetLength(1);
      double[] colData = new double[height];
      for (int col = 0; col < length; col++) {
        for (int row = 0; row < height; row++) {
          colData[row] = matrix[row, col];
        }
        Transform(colData);
        for (int row = 0; row < height; row++) {
          matrix[row, col] = colData[row];
        }
      }
    }
  }
}///:~
```

Get things right...

The cardinal rule of performance programming is to first get the system operating properly and then worry about performance. The second rule is to always use a profiler to measure where your problems are, never go with a guess. In an object-oriented design, after discovering a hotspot, you should always break the problem out into an abstract data type (an interface) if it is not already. This will allow you to switch between different implementations over time, confirming that your performance work is accomplishing something and that it is not diverging from your correct "safe" work.

In this case, the Wavelet class uses an interface called Transform to perform the actual work:

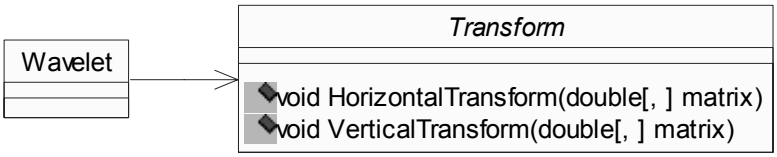


Figure 10-3: The Wavelet class relies on the Transform interface

The **Transform** interface contains two methods, each of which takes a rectangular array as a parameter and performs an in-place transformation; **HorizontalTransform( )** converts a row of values into a row containing the averages and differences of the row, and **VerticalTransform( )** performs a similar transformation on the columns of the array.

The **Wavelet** class contains two **Transform2D( )** methods, the first of which takes a rectangular array and a **Transform**. The number of steps required to perform a full wavelet transform is calculated by first determining the minimum dimension of the passed-in matrix and then using the **Math.Log( )** function to determine the base-2 magnitude of that dimension. **Math.Floor( )** rounds that magnitude down and the result is cast to the integer number of steps that will be applied to the matrix. (Thus, an array with a minimum dimension of 4 would have 2 steps; an array with 1024 would have 9.)

The constructor then calls the second constructor, which takes the same parameters as the first plus the number of times to apply the wavelet (this is a separate constructor because during debugging a single wavelet step is much easier to comprehend than a fully processed one, as Figure 10-4 illustrates)

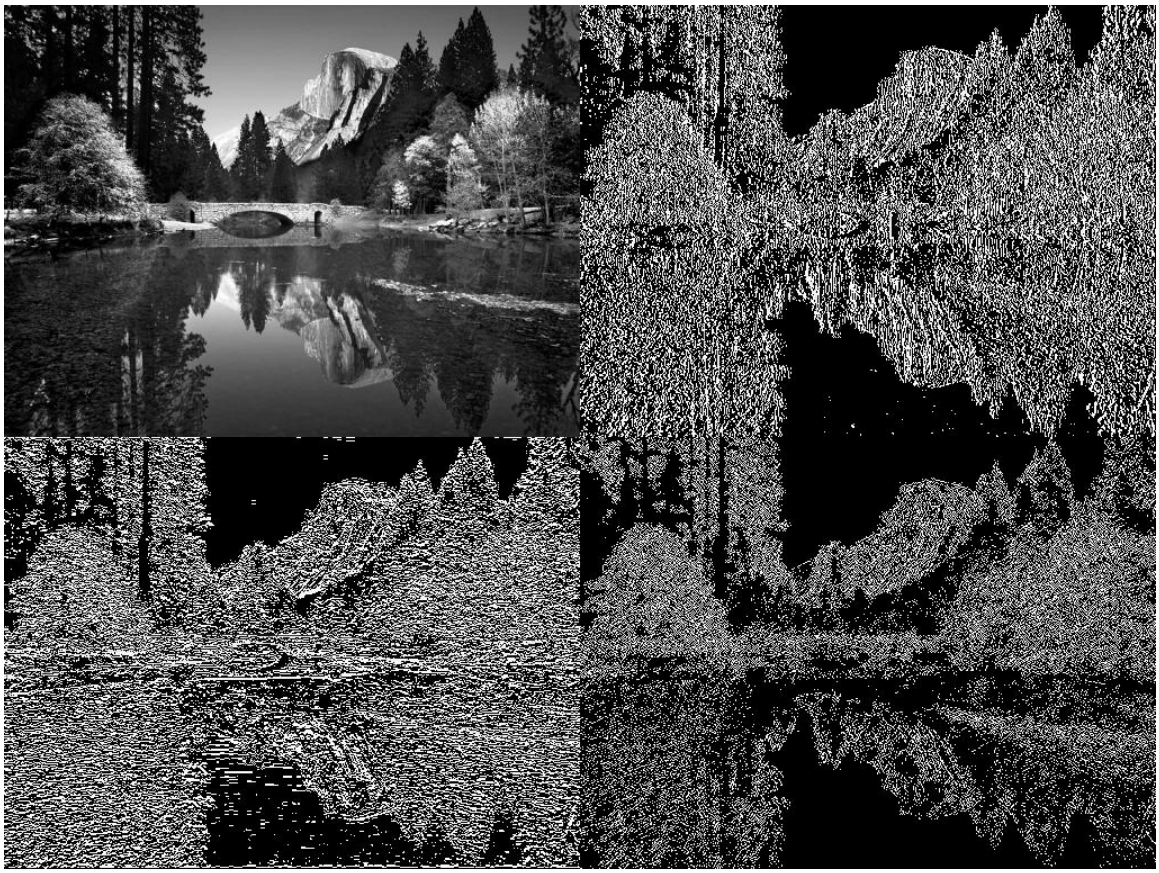


Figure 10-4: The results of one step of a Haar wavelet on a black-and-white photo

The **Transform2D( )** method iterates **steps** times over the matrix, first performing a horizontal transform and then performing a vertical transform. Alternating between horizontal and vertical transforms is called the *nonstandard wavelet decomposition*. The *standard decomposition* performs **steps** horizontal transforms and then performs **steps** vertical transforms. With graphics anyway, the nonstandard decomposition allows for easier appreciation of the wavelet behavior; in Figure 10-4, the upper-left quadrant is a half-resolution duplicate of the original, the upper-right a map of 1-pixel horizontal features, the lower-left a similar map of vertical features, and the lower-right a complete map of 1-pixel features. When the result is transformed again and again, the result has many interesting features, including being highly compressible with both lossless and lossy techniques.

The **TestSpeed( )** method in **Wavelet** creates a 4,000,000-element square array, fills it with random doubles, and then calculates and prints the time necessary to perform a full wavelet transform on the result. The **Main( )** method calls this **TestSpeed( )** method 10 times in order to ensure that any transient operating system events don't skew the results. This first version of the code calls **TestSpeed( )** with a **SafeTransform** – get things right and then get them fast.

The **SafeTransform** class has a private **Transform( )** method which takes a one-dimensional array of doubles. It creates two arrays, **avg** and **diff** of half the width of the original. The first loop in **Transform( )** moves across the source array, reading value pairs. It calculates and places these pairs' average and difference in the **avg** and **diff** arrays. After this loop finished, the values in **avg** are copied to the first half of the input array and the values in **diff** to the second half. After **Transform( )** finishes, the input array now contains the values of a one-step, one-dimensional Haar transformation. (Note that the transform is fully reversible — the original data can be restored by first adding and then subtracting a **diff** value to a corresponding **avg** value.)

SafeTransform.HorizontalTransform( ) determines the height of the passed-in matrix and copies the values of each row into a one-dimensional array of doubles called **row**. Then the code calls the previously described **Transform( )** method and copies the result back into the original two-dimensional matrix. When **HorizontalTransform( )** is finished, the input matrix as a whole now contains a one-step, horizontal Haar transformation.

SafeTransform.VerticalTransform( ) uses a similar set of loops as **HorizontalTransform( )**, but instead of copying rows from the input matrix, it copies the values in a column into a double array called **colData**, transforms that with **Transform( )**, and copies the result back into the input matrix. When this finishes, control returns to **Wavelet.Transform2D( )**, and one step of the wavelet decomposition has been performed.

… then get them fast

Running this through a profiler (we used Intel's vTune) shows that a lot of time is spent in the **HorizontalTransform( )** and **VerticalTransform( )** methods in addition to the **Transform( )** method itself. So, let's try to improve all three by using unsafe code:

```
//:c10:UnsafeTransform.cs
//Compile with:
// csc /unsafe FastBitmapper1.cs UnsafeTransform.cs
//and, in FastBitmapper1.cs, uncomment call to:
```

```
//TestSpeed(new UnsafeTransform());
using FastBitmapper;

internal class UnsafeTransform : Transform {
  unsafe private void Transform(double* array,
    int length) {
    //Console.WriteLine("UnsafeTransform({0}, {1}"
    //, *array, length);
    double* pOriginalArray = array;
    int halfLength = length >> 1;
    double[] avg = new double[halfLength];
    double[] diff = new double[halfLength];
    for (int pair = 0; pair < halfLength; pair++) {
      double first = *array;
      ++array;
      double next = *array;
      ++array;
      avg[pair] = (first + next) / 2;
      diff[pair] = avg[pair] - first;
    }
    for (int pair = 0; pair < halfLength; pair++) {
      pOriginalArray[pair] = avg[pair];
      pOriginalArray[pair + halfLength] = diff[pair];
    }
  }

  unsafe public void HorizontalTransform(
    double[,] matrix) {
    int height = matrix.GetLength(0);
    int width = matrix.GetLength(1);
    fixed(double* pMatrix = matrix) {
      double* pOffset = pMatrix;
      for (int row = 0; row < height; row++) {
        Transform(pOffset, width);
        pOffset += width;
      }
    }
  }

  unsafe public void VerticalTransform(
    double[,] matrix) {
```

```
    fixed(double* pMatrix = matrix) {
      int height = matrix.GetLength(0);
      int length = matrix.GetLength(1);
      double[] colData = new double[height];
      for (int col = 0; col < length; col++) {
        for (int row = 0; row < height; row++) {
          colData[row] = pMatrix[col + length * row];
        }
        fixed(double* pColData = colData) {
          Transform(pColData, height);
        }
        for (int row = 0; row < height; row++) {
          pMatrix[col + length * row] = colData[row];
        }
      }
    }
  }
}///:~
```

First, notice that **UnsafeTransform** has the same structure as **SafeTransform**, a private **Transform( )** function in addition to the public methods which implement **Transform**. This is by no means necessary, but it's a good starting place for optimization.

UnsafeTransform.Transform( ) has a signature unlike any C# signature discussed before: **unsafe private void Transform(double* array, int length)**. When a method is declared unsafe, C# allows a new type of variable, called a pointer. A pointer contains a memory address at which a value of the specified type is located. So the variable **array** contains not a double value such as 0.2 or 234.28, but a memory location someplace in the runtime, the contents of which are interpreted as a double. Adding 1 to **array** does not change it to 1.2 or 235.28 but rather changes the memory location to point to the next location in memory that's big enough to hold a **double**. Such "pointer arithmetic" is marginally more efficient than using a C# array, but even small differences add up when applied to a 4,000,000 item array!

The first line in **UnsafeTransform.Transform( )** initializes another pointer variable **pOriginalArray** with the original value in **array**, whose value is going to change. The declaration of the **avg** and **diff** arrays and the first loop are identical with what was done in **SafeTransform.Transform( )**, except that this time we use the value of the passed-in **length** variable to calculate the value of **halfLength** (in **SafeTransform.Transform( )**, we used the **Length** property of the passed-

in array, but pointers don't have such a property, so we need the extra parameter). The next lines, though, are quite different:

```
double first = *array;
++array;
double next = *array;
++array;
```

When applied to a pointer variable, the * operator retrieves the value that is stored at that address (the mnemonic is "star = stored"). So the **first** double is assigned the value of the double at **array**'s address value. Then, we use pointer arithmetic on **array** so that it skips over a double's worth of memory, read the value there as a double and assign it to **next** and increment **array** again. The values of **avg** and **diff** are calculated just as they were in **SafeTransform.Transform( )**.

So the big difference in this loop is that instead of indexing in to an array of **double**s of a certain length, we've incremented a pointer to **double**s **length** times, and interpreted the memory of where we were pointing at as a series of **double**s. There's been no bounds or type checking on the value of our **array** pointer, so if this method were called with either **array** set incorrectly or with a wrong **length**, this loop would blithely read whatever it happened to be pointing at.

Such a situation might be hard to track down, but the final loop in **Unsafe.Transform( )** would probably not go undetected. A feature of pointers is that you can use array notation to indicate an offset in memory. Thus, in this loop, we write back into the region of memory at **pOriginalArray** large enough to contain **length** doubles. Writing into an invalid region of memory is a pretty sure way to cause a crash. So it behooves us to make sure that **Unsafe.Transform( )** is only called properly.

Unsafe.HorizontalTransform( ) takes a two-dimensional rectangular array of doubles called **matrix**. Before calling **Unsafe.Transform( )**, which takes a pointer to a double, the **matrix** must be "pinned" in memory. The .NET garbage collector is normally free to move objects about, because the garbage collector has the necessary data to determine every reference to that object (indeed, tracking those references is the very essence of garbage collection!). But when a pointer is involved, it's not safe to move references; in our case, the loops in **Transform** both read and write a large block of memory based on the original passed-in address.

The line **fixed(double* pMatrix = matrix)** pins the rectangular array **matrix** in memory and initializes a pointer to the beginning of that memory. Pointers initialized in a **fixed** declaration are read-only and for the purposes of pointer

arithmetic, we need the next line to declare another pointer variable **pOffset** and initialize it to the value of **pMatrix**.

Notice that unlike **SafeTransform.HorizontalTransform( )**, we do not have a temporary one-dimensional **row** array which we load before calling **Transform( )** and copy from after. Instead, the main loop in **HorizontalTransform( )** calls **Transform( )** with its pointer of **pOffset** and its length set to the previously calculated width of the input **matrix**. Then, we use pointer arithmetic to jump **width** worth of **double**s in memory. In this way, we are exploiting the fact that we know that a rectangular array is, behind-the-scenes, a contiguous chunk of memory. The line **pOffset += width;** is significantly faster than the 8 lines of safe code it replaces.

In **UnsafeTransform.VerticalTransform( )**, though, no similar shortcut comes to mind and the code is virtually identical to that in **SafeTransform.VerticalTransform( )** except that we still need to pin **matrix** in order to get the **pMatrix** pointer to pass to **Transform( )**.

If we go back to Wavelet.Main() and uncomment the line that calls **TestSpeed( )** with a **new UnsafeTransform( )**, we're almost ready to go. However, the C# compiler requires a special flag in order to compile source that contains unsafe code. On the command-line, this flag is /unsafe , while in Visual Studio .NET, the option is found by right-clicking on the Project in the Solution Explorer and choosing Properties / Configuration Properties / Build and setting "Allow unsafe code blocks" to **true**.

On my machines, **UnsafeTransform** runs about 50% faster than **SafeTransform** in debugging mode, and is about 20% superior when optimizations are turned on. Hardly the stuff of legend, but in a core algorithm, perhaps worth the effort.

There's only one problem. This managed code implementation runs 40% faster than **UnsafeTransform**! Can you reason why?:

```
//:c10:InPlace.cs
//Compile with:
//csc /reference:FastBitmapper1.exe InPlace.cs
//Add timing code to FastBitmapper to test speed.
using FastBitmapper;

internal class InPlace : Transform {
  int length;
  int height;
```

```
int halfLength;
int halfHeight;
//Half the length of longer dimension
double[] diff = null;

private void LazyInit(double[,] matrix) {
  height = matrix.GetLength(0);
  length = matrix.GetLength(1);
  halfLength = length >> 1;
  halfHeight = height >> 1;
  if (halfHeight < halfLength) {
    diff = new double[halfLength];
  } else {
    diff = new double[halfHeight];
  }
}

public void HorizontalTransform(double[,] matrix) {
  if (diff == null) {
    LazyInit(matrix);
  }
  for (int i = 0; i < height; i++) {
    HTransform(matrix, i);
  }

}

public void VerticalTransform(double[,] matrix) {
  if (diff == null) {
    LazyInit(matrix);
  }

  for (int col = 0; col < length; col++) {
    VTransform(matrix, col);
  }
}

private void HTransform(double[,] matrix, int row) {
  for (int pair = 0; pair < halfLength; pair++) {
    double first = matrix[row, pair * 2];
    double next = matrix[row, pair * 2 + 1];
```

```
      double avg = (first + next) / 2;
      matrix[row, pair * 2] = avg;
      diff[pair] = avg - first;
    }
    for (int pair = 0; pair < halfLength; pair++) {
      matrix[row, pair + halfLength] = diff[pair];
    }
  }

  private void VTransform(double[,] matrix, int col) {
    for (int pair = 0; pair < halfHeight; pair++) {
      double first = matrix[pair * 2, col];
      double next = matrix[pair * 2 + 1, col];
      double avg = (first + next) / 2;
      matrix[pair * 2, col] = avg;
      diff[pair] = avg - first;
    }
    for (int pair = 0; pair < halfHeight; pair++) {
      matrix[pair + halfHeight, col] = diff[pair];
    }
  }
}///:~
```

InPlace removes loops and allocations of temporary objects (like the **avg** and **diff** arrays) at the cost of clarity. In **SafeTransform**, the Haar algorithm of repeated averaging and differencing is pretty easy to follow just from the code; a first-time reader of **InPlace** might not intuit, for instance, that the contents of the **diff** array are strictly for temporary storage.

Notice that both **HorizontalTransform( )** and **VerticalTransform( )** check to see if **diff** is null and call **LazyInit( )** if it is not. Some might say "Well, we know that **HorizontalTransform( )** is called first, so the check in **VerticalTransform( )** is superfluous." But if we were to remove the check from **VerticalTransform( )**, we would be changing the design contract of the **Transform( )** interface to include "You must call **HorizontalTransform( )** before calling **VerticalTransform( )**."

Changing a design contract is not the end of the world, but it should always be given some thought. When a contract requires that method **A( )** be called before method **B( )**, the two methods are said to be "sequence coupled." Sequence coupling is usually acceptable (unlike, say, "internal data coupling" where one class directly writes to another class's variables without using properties or methods to

access the variables). Given that the check in **VerticalTransform( )** is not within a loop, changing the contract doesn't seem worth what will certainly be an unmeasurably small difference in performance.

Array summary

To summarize what you've seen so far, the first and easiest choice to hold a group of objects of a known size is an array. Arrays are also the natural data structure to use if the way you wish to access the data is by a simple index, or if the data is naturally "rectangular" in its form. In the remainder of this chapter we'll look at the more general case, when you don't know at the time you're writing the program how many objects you're going to need, or if you need a more sophisticated way to store your objects. C# provides a library of *collection classes* to solve this problem, the basic types of which are **IList** and **IDictionary**. You can solve a surprising number of problems using these tools!

Among their other characteristics, the C# collection classes will automatically resize themselves. So, unlike arrays, you can put in any number of objects and you don't need to worry about how big to make the container while you're writing the program.

Cloning

When you copy an array of objects, you get a copy of the references to the single heap-based object (see Page 50). To revisit the metaphor we used in Chapter 2, you get a new set of remote controls for your existing television, not a new television. But what if you *want* a new television in addition to a new set of remote controls? This is the dilemma of cloning. Why a dilemma? Because cloning introduces the problem of shallow versus deep copying.

When you copy just the references, you have a *shallow* copy. Shallow copies are, naturally, simple and fast. If you have come this far in the book and are comfortable with the difference between reference and value types, shallow copies should not require any extra explanation. But in many situations, not just when it comes to arrays or collection classes, there are times when you'd like to have a *deep* copy, one in which you get a new version of the object and all its related objects with all the values of the fields and properties set to the value of the original object. In the world of objects, deep copies are often called *clone*s.

Your first take on cloning might be to create a new object and instantiate its fields to the values of the original:

```
//:c10:SimpleClone.cs
```

```
//Simple objects are easy to clone
using System;

enum Upholstery{ leather, fabric };
enum Color { mauve, taupe, ecru };

class Couch{
  Upholstery covering;
  Color aColor;

  Couch Clone(){
    Couch clone = new Couch();
    clone.covering = this.covering;
    clone.aColor = this.aColor;
    return clone;
  }

  public override string ToString(){
    return String.Format("Couch is {0} {1}",
      aColor, covering);
  }

  public static void Main(){
    Couch firstCouch = new Couch();
    firstCouch.covering = Upholstery.leather;
    firstCouch.aColor = Color.mauve;

    Couch secondCouch = firstCouch.Clone();
    bool areTheSame = firstCouch == secondCouch;
    Console.WriteLine("{0} == {1}: {2}",
      firstCouch, secondCouch, areTheSame);
  }
}///:~
```

The **Couch** class declares a method **Clone( )** that creates a new **Couch** on the heap and copies the field values. Although the cloned **Couch** has identical values as the original, **areTheSame** is **false**, since they are in fact different objects. Cloning objects whose fields are all value types can indeed as simple as this, but what if your objects contains a field that is supposed to be unique per instance or references to other objects?

For instance, we have used this idiom in this book to give similar objects a unique id:

```
static int idCounter = 0;
int id = idCounter++;
```

If we were to have such a field in **Couch**, should the clone have a unique **id** or should it have a copy of the value of the **firstCouch**'s **id**? There's no "correct" answer to that question and its more general extension to complex objects that have relationships with other objects – if you want a new television, does that also mean you want a new television stand (probably), a new electrical circuit for the house (probably not), a new television transmitter (definitely not), etc.?

This is very similar to the challenge of initializing an object to a consistent state, as discussed in Chapter 5. Just as there is no single way to know how many and what type of other objects an object must create in its constructor, there is no way to know how many other and what type of other objects must be created in the cloning process. As with initialization, the use of inheritance can shield the client programmer from the complexity of the process, but unlike constructors, which all classes must have and which can always be counted on to ultimately call the **Object( )** constructor, cloning requires you to implement an interface.

The **ICloneable** interface has one method: **object Clone( )**. On top of that, the **Object** class has a method called **MemberwiseClone( )** that performs a very fast bit-by-bit shallow copy of the object, so we can rewrite the previous example this way:

```
//:c10:SimpleCloneable.cs
//Implementing ICloneable
using System;

enum Upholstery{ leather, fabric };
enum Color { mauve, taupe, ecru };

class Couch : ICloneable{
  Upholstery covering;
  Color aColor;

  public object Clone(){
    return MemberwiseClone();
  }

  public override string ToString(){
```

```
    return String.Format("Couch is {0} {1}",
      aColor, covering);
  }

  public static void Main(){
    Couch firstCouch = new Couch();
    firstCouch.covering = Upholstery.leather;
    firstCouch.aColor = Color.mauve;

    Couch secondCouch = (Couch) firstCouch.Clone();
    bool areTheSame = firstCouch == secondCouch;
    Console.WriteLine("{0} == {1}: {2}",
      firstCouch, secondCouch, areTheSame);
  }
}///:~
```

The output is the same as the previous and the effort may not seem worth it for our simple couch. But in a more complex situation, the **Clone( )** method comes into its own:

```
//:c10:ComplexClone.cs
//A slightly more complex object
using System;
using System.Text;

enum Upholstery { leather, fabric };
enum Color { mauve, taupe, ecru };

class Furniture {
  protected static int idCounter = 0;
  protected int id = idCounter++;

  protected Furniture(){
    Console.WriteLine("Furniture {0} in construction",
      id);
  }

  protected Upholstery covering;
  protected Color aColor;
}

class Ottoman : Furniture {
```

```
  internal Ottoman(){
    Console.WriteLine("Ottoman created");
    covering = Upholstery.fabric;
    aColor = Color.ecru;
  }

  public override string ToString(){
    return String.Format("Ottoman {0} is {1} {2}",
      id, aColor, covering);
  }
}

class Couch : Furniture, ICloneable {
  Ottoman ottoman;

  protected Couch(Upholstery h, Color c){
    Console.WriteLine("Couch created");
    ottoman = new Ottoman();
    covering = h;
    aColor = c;
  }

  public object Clone(){
    Couch c = (Couch) MemberwiseClone();
    c.id = idCounter++; //Must override memberwise
    Console.WriteLine(
      "Couch {0} cloned into Couch {1}", id, c.id);
    return c;
  }

  public override string ToString(){
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("Couch {0} is {1} {2} with {3}",
      id, aColor, covering, ottoman);
    return sb.ToString();
  }

  public static void Main(){
    Couch firstCouch = new Couch(
      Upholstery.fabric, Color.ecru);
```

```
    Couch secondCouch = (Couch) firstCouch.Clone();
    bool areTheSame = firstCouch == secondCouch;
    Console.WriteLine("{0} == {1}: {2}",
      firstCouch, secondCouch, areTheSame);

    bool ottomansTheSame =
    firstCouch.ottoman == secondCouch.ottoman;
    Console.WriteLine("Ottomans the same: "
      + ottomansTheSame);
  }
}///:~
```

In the **Furniture** class, we use our **idCounter** and **id** idiom and when the **firstCouch** is constructed, it is assigned **id** 0 and the **Ottoman** it creates is assigned **id** 1. When **Couch.Clone( )** is called, it uses **MemberwiseClone( )** to duplicate its values. When you run this, you will see that because **MemberwiseClone( )** is a bit-level copy of memory as opposed to a more disciplined (but slower) constructor call, the cloning of the **firstCouch** *does not* activate the **Couch** constructor (and thereby the **Ottoman** constructor): The **id** does not change, you *do not* see "Furniture in construction," etc.

So to make the **id** in the cloned **Couch** act like we want, we have to manually perform the **idCounter++** call. Further, the **ottoman** is not cloned, which is the desire we want (two ecru fabric couches sharing a single ottoman is *the* look in New York nowadays).

The **ICloneable** interface gives you an initialization mechanism that is an alternate to the constructor, one which allows you to create a combination of shallow and deep copy semantics that are appropriate to your needs. **MemberwiseClone( )** is a very fast way to copy your objects, but as it bypasses the more common initialization mechanisms, its behavior can be surprising.

Introduction to data structures

The discussion of cloning touched upon the complexities that arise when you move into a world of complex relationships between objects. Container classes are one of the most powerful tools for raw development because they provide an entry into the world of data structure programming. An interesting fact of programming is that the hardest challenges often boil down to selecting a data structure and applying a handful of simple operations to it. Object orientation makes it trivial to create data

structures that work with abstract data types (i.e., a collection class is written to work with type **object** and thereby works with everything).

The .NET System.Collections namespace takes the issue of "holding your objects" and divides it into two distinct concepts:

1. IList: a group of individual elements, often with some rule applied to them. An IList must hold the elements in a particular sequence, and a Set cannot have any duplicate elements. (Note that the .NET Framework does not supply either a set, which is a Collection without duplicates, or a bag, which is an unordered Collection.)

2. IDictionary: a group of key-value object pairs (also called Maps). Strictly speaking, an IDictionary contains DictionaryEntry structures, which themselves contain the two references (in the Key and Value properties). The Key property cannot be null and must be unique, while the Value entry may be null or may point to a previously referenced object. You can access any of these parts of the IDictionary structure – you can get the DictionaryEntry values, the set of Keys or the collection of Values. Dictionaries, like arrays, can easily be expanded to multiple dimensions without adding new concepts: you simply make an IDictionary whose values are of type IDictionary (and the values of those dictionaries can be dictionaries, etc.)

Queues and stacks

For scheduling problems and other programs that need to deal with elements in order, but which when done discard or hand-off the elements to other components, you'll want to conside a queue or a stack.

A queue is a data structure which works like a line in a bank; the first to arrive is the first to be served.

A stack is often compared to a cafeteria plate-dispenser – the last object to be added is the first to be accessed. This example uses this metaphor to show the basic functions of a queue and a stack:

```
//:c10:QueueAndStack.cs
//Demonstrate time-of-arrival data structures
using System;
using System.Collections;

class Customer {
```

```
  string name;
  public string Name{
    get{ return name;}
  }

  PlateDispenser p;

  internal Customer(String name, PlateDispenser p){
    this.name = name;
    this.p = p;
  }

  internal void GetPlate(){
    string plate = p.GetPlate();
    Console.WriteLine(
      name + " got " + plate);
  }
}

class PlateDispenser {
  Stack dispenser = new Stack();
  internal void Fill(int iToPush){
    for (int i = 0; i < iToPush; i++) {
      string p = "Plate #" + i;
      Console.WriteLine("Loading " + p);
      dispenser.Push(p);
    }
  }

  internal string GetPlate(){
    return(string) dispenser.Pop();
  }
}

class Teller {
  Queue line = new Queue();
  internal void EnterLine(Customer c){
    line.Enqueue(c);
  }
  internal void Checkout(){
    Customer c = (Customer) line.Dequeue();
```

```
    Console.WriteLine("Checking out: " + c.Name);
  }
}
class Cafeteria {
  PlateDispenser pd = new PlateDispenser();
  Teller t = new Teller();

  public static void Main(){
    new Cafeteria();
  }

  public Cafeteria(){
    pd.Fill(4);
    Customer[] c = new Customer[4];
    for (int i = 0; i < 4; i++) {
      c[i] = new Customer("Customer #" + i, pd);
      c[i].GetPlate();
    }
    for (int i = 0; i < 4; i++) {
      t.EnterLine(c[i]);
    }
    for (int i = 0; i < 4; i++) {
      t.Checkout();
    }
  }
}///:~
```

First, the code specifies that it will be using types from the System and System.Collection namespaces. Then, the **Customer** class has a name and a reference to a **PlateDispenser** object. These references are passed in the **Customer** constructor. Finally, **Customer.GetPlate( )** retrieves a plate from the **PlateDispenser** and prints out the name of the customer and the identifier of the plate.

The **PlateDispenser** object contains an internal reference to a **Stack**. When **PlateDispenser.Fill( )** is called, a unique string is created and **Stack.Push( )** places it on the top (or front) of the stack. Similarly, **PlateDispenser.GetPlate( )** uses **Stack.Pop( )** to get the object at the stack's top.

The **Teller** class has a reference to a **Queue** object. **Teller.EnterLine( )** calls **Queue.Enqueue( )** and **Teller.Checkout( )** calls **Queue.Dequeue( )**. Since the only objects placed in the queue are of type **Customer**, it's safe for the

reference returned by **Queue.Dequeue( )** to be cast to a **Customer**, and the name printed to the console.

Finally, the **Cafeteria** class brings it all together. It contains a **PlateDispenser** and a **Teller**. The constructor fills the plate dispenser and creates some customers, who get plates, get in line for the teller, and check out. The output looks like this:

```
Loading Plate #0
Loading Plate #1
Loading Plate #2
Loading Plate #3
Customer #0 got Plate #3
Customer #1 got Plate #2
Customer #2 got Plate #1
Customer #3 got Plate #0
Checking out: Customer #0
Checking out: Customer #1
Checking out: Customer #2
Checking out: Customer #3
```

As you can see, the order in which the plates are dispensed is the reverse of the order in which they were placed in the **PlateDispenser**'s **Stack**.

What happens if you call **Pop( )** or **Dequeue( )** on an empty collection? In both situations you'll get an **InvalidOperationException** with an explicit message that the stack or queue is empty.

Stacks and queues are just the thing for scheduling problems, but if you need to choose access on more than a time-of-arrival basis, you'll need another data structure.

ArrayList

If a numeric index is all you need, the first thing that you'll consider is an Array, of course. But if you don't know the exact number of objects that you'll need to store, consider ArrayList.

Like the other collection classes, ArrayList has some very handy static methods you can use when you want to ensure certain characteristics of the underlying collection. The static methods **ArrayList.FixedSize( )** and **ArrayList.ReadOnly( )** return their **ArrayList** arguments wrapped in specialized handles that enforce these restrictions. However, care must be taken to discard any references to the original argument to these methods, because the inner ArrayList can get around the restrictions, as this example shows:

```
//:c10:ArrayListStatics.cs
using System;
using System.Collections;

public class ArrayListStatics {
  public static void Main(){
    ArrayList al = new ArrayList();
    Random rand = new Random();
    int iToAdd = 50 + rand.Next(50);
    for (int i = 0; i < iToAdd; i++) {
      string s = "String #" + i;
      al.Add(s);
    }
    ArrayList noMore = ArrayList.FixedSize(al);
    try {
      noMore.Add("This won't work");
    } catch (Exception ex) {
      Console.WriteLine(ex);
    }
    ArrayList untouchable =
      ArrayList.ReadOnly(al);
    try {
      untouchable[0] = "This won't work";
    } catch (Exception ex) {
      Console.WriteLine(ex);
    }
    //But restrictions do not apply to original
    al[0] = "Modified";
    Console.WriteLine(
      "Untouchable[0] = " + untouchable[0]);
    int originalCount = noMore.Count;
    al.Insert(0, "Added");
    Console.WriteLine(
      "Size of noMore {0} != {1}",
      originalCount, noMore.Count);
  }
}///:~
```

While the operations on the wrapped arrays will raise **NotSupportedExceptions** (which are caught and printed to the console), a change to the original **al** ArrayList is reflected in **Untouchable** and the size of

noMore can be increased! Another interesting static method of ArrayList is **Synchronized**, which will be discussed in Chapter 16.

BitArray

A **BitArray** is used if you want to efficiently store a lot of on-off or true-false information. It's efficient only from the standpoint of size; if you're looking for efficient access, it is slightly slower than using an array of some native type.

A normal container expands as you add more elements, but with **BitArray**, you must set the **Length** property to be sufficient to hold as many as you need. The constructor to **BitArray** takes an integer which specifies the initial capacity (there are also constructors which copy from an existing **BitArray**, from an array of bools, or from the bit-values of an array of bytes or ints).

The following example shows how the **BitArray** works:

```
//:c10:Bits.cs
// Demonstration of BitSet.
using System;
using System.Collections;

public class Bits {
  static void PrintBitArray(BitArray b) {
    Console.WriteLine("bits: " + b);
    string bbits = "";
    for (int j = 0; j < b.Length ; j++)
      bbits += (b[j] ? "1" : "0");
    Console.WriteLine("bit pattern: " + bbits);
  }

  public static void Main() {
    Random rand = new Random();
    // Take the LSB of Next():
    byte bt = (byte)rand.Next();
    BitArray bb = new BitArray(8);
    for (int i = 7; i >=0; i--)
      if (((1 << i) &  bt) != 0)
        bb.Set(i, true);
      else
        bb.Set(i, false);
    Console.WriteLine("byte value: " + bt);
```

```
    PrintBitArray(bb);

    short st = (short)rand.Next();
    BitArray bs = new BitArray(16);
    for (int i = 15; i >=0; i--)
      if (((1 << i) &  st) != 0)
        bs.Set(i, true);
      else
        bs.Set(i, false);
    Console.WriteLine("short value: " + st);
    PrintBitArray(bs);

    int it = rand.Next();
    BitArray bi = new BitArray(32);
    for (int i = 31; i >=0; i--)
      if (((1 << i) &  it) != 0)
        bi.Set(i, true);
      else
        bi.Set(i, false);
    Console.WriteLine("int value: " + it);
    PrintBitArray(bi);

    // Test BitArrays that grow:
    BitArray b127 = new BitArray(64);
    //! Would throw ArgumentOutOfRangeException
    //! b127.Set(127, true);
    //Must manually expand the Length
    b127.Length = 128;
    b127.Set(127, true);
    Console.WriteLine(
      "set bit 127: " + b127);
  }
} ///:~
```

Dictionaries

Dictionaries allow you to rapidly look up a value based on a unique non-numeric key and are among the most handy of the collection classes.

Hashtable

The **Hashtable** is so commonly used that many programmers use the phrase interchangeably with the concept of a dictionary! The **Hashtable**, though, is an implementation of **IDictionary** that has all types of interesting implementation details. Before we get to those, here's a simple example of using a **Hashtable**:

```
//:c10:SimpleHash.cs
using System;
using System.Collections;

public class LarrysPets {
  static IDictionary Fill(IDictionary d) {
    d.Add("dog", "Cheyenne");
    // Non-unique key causes exception
    //! d.Add("dog", "Bette");
    d.Add("cat", "Harry");
    d.Add("goldfish", null);
    return d;
  }
  public static void Main() {
    IDictionary pets = new Hashtable();
    Fill(pets);
    foreach(DictionaryEntry pet in pets){
      Console.WriteLine(
        "Larry has a {0} named {1}",
        pet.Key, pet.Value);
    }
  }
} ///:~
```

produces output of:

```
Larry has a dog named Cheyenne
Larry has a goldfish named
Larry has a cat named Harry
```

Note that attempting to add a non-unique key to a **Hashtable** raises an **ArgumentException.** This does not mean that one cannot change the value of a Hashtable at a given key, though:

```
//:c10:ChangeHashtableValue.cs
using System;
using System.Collections;
```

```
class ChangeHashtableValue {
  public static void Main(){
    Hashtable h = new Hashtable();
    h.Add("Foo","Bar");
    Object o = h["Foo"];
    h["Foo"] = "Modified";
    Console.WriteLine("Value is: " + h["Foo"]);
    h["Baz"] = "Bozo";
  }
}///:~
```

This example shows the use of C#'s *custom indexers*. A custom indexer allows one to access an **IDictionary** using normal array notation. Although here we use only **string**s as the keys, the keys in an **IDictionary** can be of any type and can be mixed and matched as necessary.

After "Foo" is set as the key to the "Bar" value, array notation can be used to access the value, for both reading *and* writing. As shown in the last line of **Main( )**, the same array notation can be used to add new key-value pairs to the **Hashtable**.

The most interesting **Hashtable** implementation detail has to do with the calculation of the hashcode, a unique integer which "somehow" identifies the key's unique value. The hashcode is returned by **object.GetHashCode( )**, a method that needs to be fast and to return integers that are "spread out" as much as possible. Additionally, the method must always return the same value for a given object, so you can't base your hashcode on things like system time. In this example, the hashcode and the related **object.Equals( )** method are used to express the idea that the sole determinant of a circle's identity is its center and radius :

```
//:c10:OverridingHash.cs
using System;
using System.Collections;
class Circle {
  int x, y, radius;
  internal Circle(int x, int y, int radius){
    this.x = x;
    this.y = y;
    this.radius = radius;
  }

  internal Circle(int topX, int topY, int lowerX,
    int lowerY){
```

```
    this.x =  topX + (lowerX - topX) / 2;
    this.y = topY + (lowerY - topY) / 2;
    this.radius = (lowerX - topX) / 2;
  }

  public override int GetHashCode(){
    Console.WriteLine(
      "Returning {0}", x + y + radius);
    return x + y + radius;
  }

  public override bool Equals(Object o){
    if (o is Circle) {
      Circle that = (Circle) o;
      Console.WriteLine(
        "Comparing {0},{1},{2} with " +
        "{3},{4},{5}", x, y, radius,
        that.x, that.y, that.radius);
      return(this.x == that.x) &&
      (this.y == that.y) &&
      (this.radius == that.radius);
    }
    return false;
  }

  public static void Main(){
    Circle c = new Circle(15, 15, 5);
    Circle unlike = new Circle(15, 15, 6);
    Circle somewhatLike = new Circle(30, 1, 4);
    IDictionary d = new Hashtable();
    d.Add(c, "A circle");
    d.Add(unlike, "Another circle");
    try {
      Circle like = new Circle(10, 10, 20, 20);
      d.Add(like, "Just like c");
    } catch (Exception ex) {
      Console.WriteLine(ex);
    }
  }
}///:~
```

When a **Circle** is added to a **Hashtable**, the **Hashtable** calls back to **Circle.GetHashCode( )** which returns the sum of the center coordinates and radius of the circle. This is no problem for the first two circles, **c** and **unlike**, because they have different hashcodes. Circle **somewhatLike**, though, causes what is called a "hash collision" – the same hashcode is returned for two different objects (in this case, both circles' elements add up to 35). When a hash collision takes place, **Hashtable** calls **object.Equals( )** to see if the objects are, in fact, the same object. Because these two circles have different centers and radii, they can both be added to the **Hashtable**. Hash collisions seriously interfere with the efficiency of the **Hashtable**, so frequent collisions should make you revisit your hashcode algorithm.

In the try block, we create another new **Circle**, this time using an alternate constructor. When it's added to the **Hashtable**, this time there's another collision, but this time **Circle.Equals( )** reveals that yes, **c** and **like** are logically equivalent and therefore **Hashtable** throws an **ArgumentException**.

ListDictionary

If you have only a dozen or fewer objects to store in your dictionary, **ListDictionary** will have better performance than a **Hashtable**. On the other hand, on larger amounts of objects, **ListDictionary** has much, *much* worse performance and the performance of a collection class with a small number of elements is unlikely to be a hotspot in an application. It's not impossible, though! So if you've got a dictionary with a small amount of objects and it's buried in the central loop in your application, **ListDictionary** might come in handy. Otherwise, go with **Hashtable**.

SortedList

Sometimes, you need to access a Collection in two different ways: key-based lookup for one purpose, and index-based lookup for another. The SortedList provides this dual-mode capability:

```
//:c10:ShowSortedList.cs
using System;
using System.Collections;

class ShowSortedList {
  SortedList monthList = new SortedList();
  ShowSortedList(){
    monthList.Add("January", 31);
    monthList.Add("February",28.25);
```

```
    monthList.Add("March", 31);
    monthList.Add("April", 30);
    monthList.Add("May", 31);
    monthList.Add("June", 30);
    monthList.Add("July", 31);
    monthList.Add("August", 31);
    monthList.Add("September",30);
    monthList.Add("October",31);
    monthList.Add("November",30);
    monthList.Add("December",31);

    Console.WriteLine(
      "June has {0} days", monthList["June"]);
    Console.WriteLine(
      "The eighth month has {0} days",
      monthList.GetByIndex(7));
  }

  public static void Main(){
    ShowSortedList ssl = new ShowSortedList();
  }
}///:~
```

The **SortedList** can be accessed using any object of the key type, in this case **string**s. Or, **GetByIndex**() can be used to retrieve a value based on a numeric index.

String specialists

Strings are certainly the most used type for keys and values, and the .NET Framework provides a number of specialized collections that work exclusively with strings. These collections can be found in the **System.Collections.Specialized** namespace.

One key, multiple values

The **NameValueCollection** serves for those situations when you want to associate a single key **string** with multiple **string** values:

```
//:c10:Months.cs
using System;
using System.Collections.Specialized;

class Months {
```

```
  public static void Main(){
    NameValueCollection months =
    new NameValueCollection();
    months.Add("Winter", "January");
    months.Add("Winter", "February");
    months.Add("winter", "December");
    months.Add("Spring", "March");
    months.Add("Spring", "April");
    months.Add("Spring", "May");
    foreach(string key in months.AllKeys){
      Console.WriteLine("Key: " + key);
      Console.WriteLine("CSV: " +
        months[key]);
      foreach(Object value in
        months.GetValues(key)){
        Console.WriteLine(
          "\tValue: " + value);
      }
    }
  }
}///:~
```

The output of the program is shown here:

```
Key: Winter
CSV: January,February,December
        Value: January
        Value: February
        Value: December
Key: Spring
CSV: March,April,May
        Value: March
        Value: April
        Value: May
```

In a rather strange design decision, the custom indexer and the **Get( )** method return the values as a single comma-separated string rather than as an array. If you want to access the values as a string array, you have to use the **GetValues( )** method.

Customizing hashcode providers

Note that in the previous Months program, the "December" value was added for the key "winter" as opposed to "January" and "February," which used the key "Winter." In the discussion of **Hashtable**, we showed how a class could override its **GetHashCode( )** and **Equals( )** methods to control placement in a **Hashtable**. Even more customization is possible by changing the strategy that the **Hashtable** or **NameValueCollection** uses to calculate equality; this can be done by creating the dictionary with a custom **IHashCodeProvider** and **IComparer**. By default, **NameValueCollection** uses a **CaseInsensitiveHashcodeProvider** and **CaseInsensitiveComparer** to determine what fits into what slot.

This program demonstrates the creation of a custom **IComparer** and **IHashCodeProvider** to create a hashtable which stores only the last even or odd integer added (note that this is certainly "the most complicated thing that could possibly work"):

```
//:c10:EvenOdd.cs
using System;
using System.Collections;

class EvenOddComparer : IComparer {
  public int Compare(Object x, Object y){
    //Only compare integers
    if (x is Int32 == false
        || y is Int32 == false) {
      throw new ArgumentException(
        "Can't compare non-Int32's");
    }
    //Unbox inputs
    int xValue = (int) x;
    int yValue = (int) y;
    if (xValue % 2 == yValue % 2) {
      return 0;
    }
    return -1;
  }
}

class EvenOddHashCodeProvider : IHashCodeProvider {
  public int GetHashCode(Object intObj){
    //Only hash integers
```

```
    if (intObj is Int32 == false) {
      throw new ArgumentException(
        "Can't hash non-Int32's");
    }
    //Unbox input
    int x = (int) intObj;
    return x % 2;
  }
}

class EvenOdd {
  static EvenOddComparer c =
    new EvenOddComparer();
  static EvenOddHashCodeProvider p =
    new EvenOddHashCodeProvider();
  //Hashtable keys
  static readonly int EVEN_KEY = 2;
  static readonly int ODD_KEY = 3;
  //Custom IComparer & IHashCodeProvider strategies
  Hashtable evenOdd = new Hashtable(p, c);

  public void Test(){
    evenOdd[EVEN_KEY] = 2;
    evenOdd[ODD_KEY] = 3;
    evenOdd[EVEN_KEY] = 4;

    Console.WriteLine(
      "The last even number added was: " +
      evenOdd[EVEN_KEY]);
    Console.WriteLine(
      "The last odd number added was: " +
      evenOdd[ODD_KEY]);
  }

  public static void Main(){
    EvenOdd eo = new EvenOdd();
    eo.Test();
  }
}///:~
```

String specialists: StringCollection **and** StringDictionary

If you only want to store strings, **StringCollection** and **StringDictionary** are marginally more efficient than their generic counterparts. A **StringCollection** implements **IList** and **StringDictionary** naturally implements **IDictionary**. Both the keys and values in **StringDictionary** must be strings, and the keys are case-insensitive and stored in lower-case form. Here's a dramatically abridged dictionary program:

```
//:c10:WebstersAbridged.cs
using System;
using System.Collections.Specialized;

class WebstersAbridged {
  static StringDictionary sd =
  new StringDictionary();

  static WebstersAbridged(){
    sd["aam"] =
    "A measure of liquids among the Dutch";
    sd["zythum"] =
    "Malt beverage brewed by ancient Egyptians";
  }

  public static void Main(string[] args){
    foreach(string arg in args){
      if (sd.ContainsKey(arg)) {
        Console.WriteLine("{0}: {1}", arg, sd[arg]);
      } else {
        Console.WriteLine(
          "{0} : I don't know that word", arg);
      }
    }
  }
}///:~
```

The program iterates over the command-line arguments and either returns the definition or admits defeat. Because the **StringDictionary** is case-insensitive, this program is highly useful even when the CAPS LOCK key on the keyboard is left turned on.

Container disadvantage: unknown type

Aside from **StringCollection** and **StringDictionary**, .NET's collection classes have the "disadvantage" of obscuring type information when you put an object into a container. This happens because the programmer of that container class had no idea what specific type you wanted to put in the container, and making the container hold only your type would prevent it from being a general-purpose tool. So instead, the container holds references to **object**, which is the root of all the classes so it holds any type. This is a great solution, except:

1. Since the type information is obscured when you put an object reference into a container, there's no restriction on the type of object that can be put into your container, even if you mean it to hold only, say, cats. Someone could just as easily put a dog into the container.

2. Since the type information is obscured, the only thing the container knows that it holds is a reference to an **object**. You must perform a cast to the correct type before you use it.

On the up side, C# won't let you *misuse* the objects that you put into a container. If you throw a dog into a container of cats and then try to treat everything in the container as a cat, you'll get a run-time exception when you pull the dog reference out of the cat container and try to cast it to a cat.

Here's an example using the basic workhorse container, **ArrayList**. First, **Cat** and **Dog** classes are created:

```
//:c10:Cat.cs
using System;

namespace pets{
  public class Cat {
    private int catNumber;
    internal Cat(int i) { catNumber = i;}
    internal void Print() {
      Console.WriteLine(
        "Cat #" + catNumber);
    }
  }
} ///:~
```

```
//:c10:Dog.cs
using System;

namespace pets{
  public class Dog {
    private int dogNumber;
    internal Dog(int i) { dogNumber = i;}
    internal void Print() {
      Console.WriteLine(
        "Dog #" + dogNumber);
    }
  }
} ///:~
```

Cats and **Dog**s are placed into the container, then pulled out:

```
//:c10:CatsAndDogs.cs
// Compile with: csc Cat.cs Dog.cs CatsAndDogs.cs
// Simple container example.
using System;
using System.Collections;

namespace pets{
  public class CatsAndDogs {
    public static void Main() {
      ArrayList cats = new ArrayList();
      for (int i = 0; i < 7; i++)
        cats.Add(new Cat(i));
      // Not a problem to add a dog to cats:
      cats.Add(new Dog(7));
      for (int i = 0; i < cats.Count; i++)
        ((Cat)cats[i]).Print();
      // Dog is detected only at run-time
    }
  }
}///:~
```

The classes **Cat** and **Dog** are distinct—they have nothing in common except that they are **object**s. (If you don't explicitly say what class you're inheriting from, you automatically inherit from **object**.) Since **ArrayList** holds **object**s, you can not only put **Cat** objects into this container using the **ArrayList** method **Add( )**, but you can also add **Dog** objects without complaint at either compile-time or run-time. When you go to fetch out what you think are **Cat** objects using the **ArrayList**

indexer, you get back a reference to an **object**. Since the intent was that **cats** should only contain felines, there is no check made before casting the returned value to a **Cat**. Since we want to call the **Print( )** method of **Cat**, we have to force the evaluation of the cast to happen first, so we surround the expression in parentheses before calling **Print( )**. At run-time, though, when the loop tries to cast the **Dog** object to a **Cat**, it throws a **ClassCastException**.

This is more than just an annoyance. It's something that can create difficult-to-find bugs. If one part (or several parts) of a program inserts objects into a container, and you discover only in a separate part of the program through an exception that a bad object was placed in the container, then you must find out where the bad insert occurred. On the upside, it's convenient to start with some standardized container classes for programming, despite the scarcity and awkwardness.

Using CollectionBase **to make type-conscious collections**

As mentioned at the beginning of the chapter, the .NET runtime will eventually natively support typed collections. In the meantime, there's **CollectionBase**, an abstract **IList** that can be used as the basis for writing a strongly typed collection. To implement a typed **IList**, one starts by creating a new type that inherits from CollectionBase. Then, one has to implement **ICollection.CopyTo( ), IList.Add( ), IList.Contains( ), IList.IndexOf( ), IList.Insert( ),** and **IList.Remove( )** with type-specific signatures. The code is straightforward; the **CollectionBase.List** property is initialized in the base-class constructor and all your code has to do is pass your strongly-typed arguments on to **CollectionBase.List**s **object**-accepting methods and casting the **List**s **object**-returning methods to be more strongly typed:

```
//:c10:CatList.cs
//Compile with:
//csc Cat.cs Dog.cs CatList.cs
//An listthat contains only Cats

using System;
using System.Collections;

namespace pets{
  class CatList : CollectionBase {
    public Cat this[int index]{
      get{ return(Cat) List[index];}
      set{ List[index] = value;}
```

```
    }

    public int Add(Cat feline){
      return List.Add(feline);
    }

    public void Insert(int index, Cat feline){
      List.Insert(index, feline);
    }

    public int IndexOf(Cat feline){
      return List.IndexOf(feline);
    }

    public bool Contains(Cat feline){
      return List.Contains(feline);
    }

    public void Remove(Cat feline){
      List.Remove(feline);
    }

    public void CopyTo(Cat[] array, int index){
      List.CopyTo(array, index);
    }

    public static void Main(){
      CatList cl = new CatList();
      for (int i = 0; i < 3; i++) {
        cl.Add(new Cat(i));
      }
      //! Can't Add(dog);
      //! cl.Add(new Dog(4));
    }
  }
}///:~
```

Note that if **CatList** had inherited directly from **ArrayList**, the methods that take references to **Cat**s, such as **Add(Cat)** would simply overload (not override) the **object**-accepting methods (e.g., **Add(object)** would still be available). Thus, the **CatList** becomes a *surrogate* to the **ArrayList**, performing some activities before passing on the responsibility (see *Thinking in Patterns with Java*).

Because a **CatList** will accept only a **Cat**, the line:

```
cl.add(new Dog(4));
```

will generate an error message *at compile-time*. This approach, while more tedious from a coding standpoint, will tell you immediately if you're using a type improperly.

Note that no cast is necessary when using **Get( )** or the custom indexer —it's always a **Cat**.

IEnumerator**s**

In any container class, you must have a way to put things in and a way to get things out. After all, that's the primary job of a container—to hold things. In the **ArrayList**, **Add( )** and **Get( )** are one set of ways to insert and retrieve objects. **ArrayList** is quite flexible—you can select anything at any time, and select multiple elements at once using different indexes.

If you want to start thinking at a higher level, there's a drawback: you need to know the exact type of the container in order to use it. This might not seem bad at first, but what if you start out using an **ArrayList**, and later on in your program you decide that because of the way you are using the container, you'd like to switch your code to use a typed collection descending from **CollectionBase**? Or suppose you'd like to write a piece of generic code that doesn't know or care what type of container it's working with, so that it could be used on different types of containers without rewriting that code?

The concept of an *enumerator* (or *iterator*) can be used to achieve this abstraction. An enumerator is an object whose job is to move through a sequence of objects and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence. In addition, an enumerator is usually what's called a "light-weight" object: one that's cheap to create. For that reason, you'll often find seemingly strange constraints for enumerators; for example, some iterators can move in only one direction.

The .NET **IEnumerator** interface is an example of these kinds of constraints. There's not much you can do with one except:

1. Ask a collection (or any other type that implements IEnumerable) to hand you an **IEnumerator** using a method called **GetEnumerator( )**. This **IEnumerator** will be ready to move to the first element in the sequence on your first call to its **MoveNext( )** method.

2. Get the current object in the sequence with the **Current** property.

3. Attempt to move to the next object in sequence with the **MoveNext( )** method. If the enumerator has reached the end of the sequence, this method returns **false**.

4. Reset to its initial state, which is *prior* to the first element (i.e., you must call **MoveNext( )** once before reading the **Current** property).

That's all. It's a simple implementation of an iterator, but still powerful. To see how it works, let's revisit the **CatsAndDogs.cs** program from earlier in this chapter. In the following modified version, we've removed the errant dog and use an **IEnumerator** to iterate over the lists contents:

```
//:c10:CatsAndDogs2.cs
//Compile with:
//csc Cat.cs Dog.cs CatsAndDogs2.cs
// Using an explicit IEnumerator
using System;
using System.Collections;

namespace pets{
  public class CatsAndDogs {
    public static void Main() {
      ArrayList cats = new ArrayList();
      for (int i = 0; i < 7; i++)
        cats.Add(new Cat(i));
      IEnumerator e = cats.GetEnumerator();
      while (e.MoveNext() != false) {
        Object c = e.Current;
        ((Cat) c).Print();
      }
    }
  }
}///:~
```

You can see that the last few lines now use an **IEnumerator** to step through the sequence instead of a **for** loop. With the **IEnumerator**, you don't need to worry about the number of elements in the container.

Behind the scenes, C#'s **foreach( )** blocks do an even better job of iterating over an **IEnumerable** type, since the **foreach** attempts to cast the **object** reference

returned from the enumerator to a specific type. It doesn't make a lot of sense to use an **IEnumerator** when you can use:

```
foreach(Cat c in cats){
   c.Print();
}
```

Custom indexers

Previously, we saw how **IDictionary** types allow one to use index notation to access key-value pairs using non-numeric indices. This is done by using operator overloading to create a custom indexer.

The type-safe **CatList** collection shown in the discussion of **CollectionBase** showed a custom indexer that took a numeric index, but returned a **Cat** instead of an **object** reference. You can manipulate both index and return types in a custom indexer, just as with any other C# method.

For instance, imagine a maze which consists of rooms connected by corridors in the rooms' walls. In such a situation, it might make sense to have the corridors be indexed by direction in the room:

```
//:c10:Room.cs
//Not intended for a stand-alone compile
using System;
using System.Collections;

namespace Labyrinth{
  enum Direction {
    north, south, east, west
  };

  class Room {
    static int counter = 0;
    protected string name;
    internal string Name{
      get{ return name;}
    }

    public override string ToString(){
      return this.Name;
    }
    internal Room(){
```

```
      name = "Room #:" + counter++;
    }
    Corridor[] c = new Corridor[
      Enum.GetValues(typeof(Direction)).Length];

    public Corridor this[Direction d]{
      get { return c[(int) d];}
      set { c[(int) d] = value;}
    }
  }

  class RegenSpot : Room {
    internal RegenSpot() : base(){
      name = "Regen Spot";
    }
  }

  class PowerUp : Room {
    internal PowerUp() : base(){
      name = "Power Up";
    }
  }
}///:~ (Example continues with Corridor.cs)
```

First, we declare a **Direction** enumeration corresponding to the cardinal points. Then we declare a Room class with an internal static counter used to give each room a unique name. The name is made into a property and is also used to override the **ToString( )** method to be a little more specific as to the exact room we're dealing with.

Every **Room** has an array called **c** that will hold references to its outbound **Corridors**. Although it would be shorter to just declare this array as size 4 (since we know that's how many **Direction**s there are), instead we use a facility we've not yet discussed.

The static method **Enum.GetValues( )** converts an enumeration into an array, whose **Length** property can be read to tell us how many values there are in the enumeration. **Enum.GetValues( )** takes as its parameter, though, a **Type** object. The general utility of **Type** objects will become more apparent in Chapter 13's discussion of reflection, but for the moment suffice it to say that the **Type** object makes the structure of the type available for programmatic use; in this case, with the **Type** object that represents the **Direction** enumeration, we can determine

that there are just 4 specified directions. The **Type** object of a given class can be retrieved by using the operator **typeof** and passing it the name of the class. The form **typeof(Classname)** may look like a method call but is actually a language-level operator.

Once we have transformed the **Direction** enumeration into an array, we can use the **Length** property of the resulting array to size the **Corridor** array equivalently (this way, we could accommodate octagonal rooms by simply adding values such as **NorthWest** to **Direction**).

The custom indexer is next and looks like this:

```
public Corridor this[Direction d]{
   get { return c[(int) d];}
   set { c[(int) d] = value;}
}
```

The first line begins with what looks like a normal declaration of a public Property of type **Corridor**, but the **this[Type t]** that ends the line indicates that it is an customer indexer, in this case one that takes a **Direction** value as its key and returns a **Corridor**. Since enums are value types that default to being represented by integer constants that start with zero, we can safely use the cast to **int** to create a numeric index to the **c** array of **Corridor**s. Like a Property, a custom indexer's **set** method has a hidden parameter called **value**.

That finishes up the **Room** class, but we declare two additional types to inherit from it – a **RegenSpot** and a **PowerUp**. They each differ from the base class solely in the way they set up their **Name** property.

The **Corridor** class referenced in **Room** has one duty – maintain references to two different **Rooms** and supply a **Traverse( )** function which returns whichever of the two rooms isn't passed in as an argument:

```
//:c10:Corridor.cs
// Not intended for a stand-alone compile
using System;
using System.Collections;

namespace Labyrinth{

  class Corridor {
    Room x, y;
```

```
    internal Corridor(
      Room x, Direction xWall,
      Room y, Direction yWall){
      this.x = x;
      this.y = y;
      x[xWall] = this;
      y[yWall] = this;
    }

    public Room Traverse(
      Room origin, Direction wall){
      Console.WriteLine(
        "Leaving {0} by its {1} corridor",
        origin, wall);
      if (origin == x)
        return y;
      else
        return x;
    }

    public Room Traverse(Room origin){
      Console.WriteLine(
        "Retreating from " + origin);
      if (origin == x)
        return y;
      else
        return x;
    }
  }
}///:~ (Example continues with Maze.cs)
```

The **Corridor( )** constructor uses **Room**'s custom indexers (e.g., **x[xWall] = this;**). Note that there's no problem in referring to **this** inside a constructor.

Custom enumerators & data structures

In the previous example, after a corridor is created, the **Corridor** contains a reference to both **Room**s, and both **Room**s contain a reference to the **Corridor**:

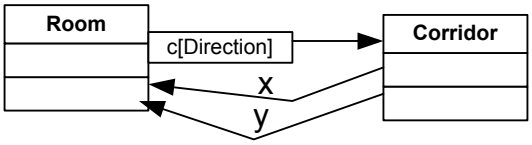


Figure 10-5: Rooms and Corridors have references to each other

While we called them rooms and corridors, what we've really got here is a *non-directed graph*. Here, "graph" is being used in its mathematical sense of "a set V of Vertices and a set E of Edges that connect elements in V." What we've designed is "non-directed" because our **Corridor**s can be traversed in either direction.

A lot of very interesting problems can be mapped into graph theory, for example: problems such as winning a game of chess, how best to pack a container, the most efficient way to schedule a bunch of jobs, and everyone's favorite, which is the cheapest route for a traveling salesperson to visit a bunch of cities. Writing a custom enumerator (or perhaps more than one, to try out different algorithms) is an elegant way to traverse a complex graph.

In this example, we create a simple Maze that consists of one **RegenSpot** and one **PowerUp**, and several normal rooms (the names are taken from videogames for which one can program "bots" – just think of them as the start and stop points):

```
//:c10:Maze.cs
//Not intended for a stand-alone compile
using System;
using System.Collections;
namespace Labyrinth{
  class Maze :IEnumerable {
    Room[] rooms;
    Room regenRoom;
    internal Room RegenSpot{
      get { return regenRoom;}
    }

    Maze(){
      regenRoom = new RegenSpot();
      rooms = new Room[]{
        new Room(), new Room(), new Room(),
        regenRoom, new Room(), new Room(),
        new PowerUp()
      };
      new Corridor(rooms[0], Direction.east,
```

```
        rooms[4], Direction.north);
      new Corridor(rooms[0], Direction.south,
        rooms[1], Direction.north);
      new Corridor(rooms[1], Direction.south,
        rooms[3], Direction.north);
      new Corridor(rooms[2], Direction.east,
        rooms[3], Direction.west);
      new Corridor(rooms[3], Direction.east,
        rooms[4], Direction.west);
      new Corridor(rooms[3], Direction.south,
        rooms[5], Direction.south);
      new Corridor(rooms[5], Direction.south,
        rooms[6], Direction.north);
    }

    public static void Main(){
      Maze m = new Maze();
      foreach(Room r in m){
        Console.WriteLine(
          "RoomRunner in " + r.Name);
      }
    }

    public IEnumerator GetEnumerator(){
      return new DepthFirst(this);
    }
  }
}///:~ (Example continues with RoomRunner.cs)
```

Class **Maze** is declared to implement the **IEnumerable** interface, which we'll use to return a customized enumerator which runs the maze. Note that for our purposes, we don't care if the enumerator visits every vertex on the graph (every room in the maze); as a matter of fact, we're probably most interested in an enumerator which visits as few vertices as possible! This is a different intention from the generic enumerators of the .NET collection classes, which of course *do* need to visit every element in the data structure.

The **Maze** contains an array **rooms** and a reference to the starting **RegenRoom**. The maze's dynamic structure is built in the **Maze( )** constructor and consists of 7 **Room**s and 7 **Corridor**s.

The **Main( )** method constructs a **Maze** and then uses a **foreach** block to show the traversal of the maze. Behind the scenes, the **foreach** block determines that **Maze** is an **IEnumerable** type and silently calls **GetEnumerator( )**.

Maze's implementation of **IEnumerable.GetEnumerator( )** is the final method in **Maze**. A new object of type **DepthFirst** (discussed shortly) is created with a reference to the current **Maze**.

There are several different ways to traverse a maze. It is probable that when writing a program to run mazes, you would want to try several different algorithms, one that rushed headlong down the first unexplored corridor, another that methodically explored all the routes from a single room, etc. However, each of these algorithms has a lot of things in common: they must all implement the **IEnumerator** interface, they all have references to the **Maze** and a current **Room**, they all begin at the regen spot and end at the power up. Really, the only way they differ is in their implementation of **IEnumerator.MoveNext( )** when they're "lost" in the maze. This is a job for the *Template Method* pattern:

```
//:c10:RoomRunner.cs
//Not intended for a stand-alone compile
using System;
using System.Collections;
namespace Labyrinth{
  abstract class RoomRunner:IEnumerator {
    Maze m;
    protected RoomRunner(Maze m){
      this.m = m;
    }

    protected Room currentRoom = null;
    public oObject Current{
      get { return currentRoom;}
    }

    public virtual void Reset(){
      currentRoom = null;
    }

    public bool MoveNext(){
      if (currentRoom == null) {
        Console.WriteLine(
          "{0} starting the maze",
```

```
            this.GetType());
          currentRoom = m.RegenSpot;
          return true;
        }
        if (currentRoom is PowerUp) {
          Console.WriteLine(
            "{0} has found PowerUp!",
            this.GetType());
          return false;
        }
        return this.ConcreteMoveNext();
      }

      protected abstract bool ConcreteMoveNext();
    }
}///:~ (Example continues with DepthFirst.cs)
```

Here, an **abstract** class called **RoomRunner** implements the methods of the **IEnumerator** interface, but leaves one tiny bit to its subclasses to implement.

The **RoomRunner( )** constructor just stores a reference to the **Maze** that creates it and initializes the **currentRoom** (exposed to the outside world as **IEnumerator**'s **Current** Property) to null. **Reset( )** also sets the **currentRoom** to null – remember that **IEnumerator.MoveNext( )** is always called once *before* the first read of the **Current** property.

The first time **RoomRunner.MoveNext( )** is called, **currentRoom** will be null. Because **RoomRunner** is an abstract type that may be implemented by many different subtypes, our console message can use **this.GetType( )** to determine the exact runtime type of **RoomRunner**. (The trick at the root of the Template Method pattern.) After printing a message to the screen announcing the **RoomRunner**'s readiness to start traversing the **Maze**, the current room is set to the **Maze**'s **RegenSpot** and the method returns true to indicate that the **IEnumerator** is at the beginning of the data structure.

Similarly, if the **currentRoom** is of type **PowerUp**, the maze running is, by our definition, complete and **MoveNext( )** returns false.

If, however, the **currentRoom** is neither **null** nor a **PowerUp** room, execution goes to **this.ConcreteMoveNext( )**. This is the template method. Just as **this.GetType( )** will return the exact runtime type, **this.ConcreteMoveNext( )** will execute the **ConcreteMoveNext( )** method of the runtime type. For this to

work, of course, **RoomRunner.ConcreteMoveNext( )** must be declared as **virtual** or, as in this case, **abstract**.

Maze.GetEnumerator( ) returned an object of type **DepthFirst**, which implements **RoomRunner**'s template method **ConcreteMoveNext( )**:

```
//:c10:DepthFirst.cs
/* Compile with:
   csc /out:RoomRunner.exe Room.cs Corridor.cs Maze.cs
   RoomRunner.cs DepthFirst.cs
*/

using System;
using System.Collections;
namespace Labyrinth{
  class DepthFirst : RoomRunner {
    public DepthFirst(Maze m) : base(m){}

    ArrayList visitedCorridors = new ArrayList();
    Corridor lastCorridor = null;

    protected override bool ConcreteMoveNext(){
      foreach(Direction d in
              Enum.GetValues(typeof(Direction))){
        if (currentRoom[d] != null) {
          Corridor c = currentRoom[d];
          if (visitedCorridors.Contains(c)
              == false) {
            visitedCorridors.Add(c);
            lastCorridor = c;
            currentRoom =
            c.Traverse(currentRoom, d);
            return true;
          }
        }
      }
      //No unvisited corridors! Retreat!
      lastCorridor.Traverse(currentRoom);
      return true;
    }
  }
}///:~
```

DepthFirst inherits from RoomBaseRunner, as its class declaration and constructor show. This particular maze runner essentially goes down the first corridor it sees that it hasn't yet gone through. If there are none it hasn't been down, it backtracks to the previous room. Uh-oh – what if it backtracks into a room and *that* room doesn't have any unvisited corridors? Looks like a defect!

But on our maze, the DepthFirst works like a champ:

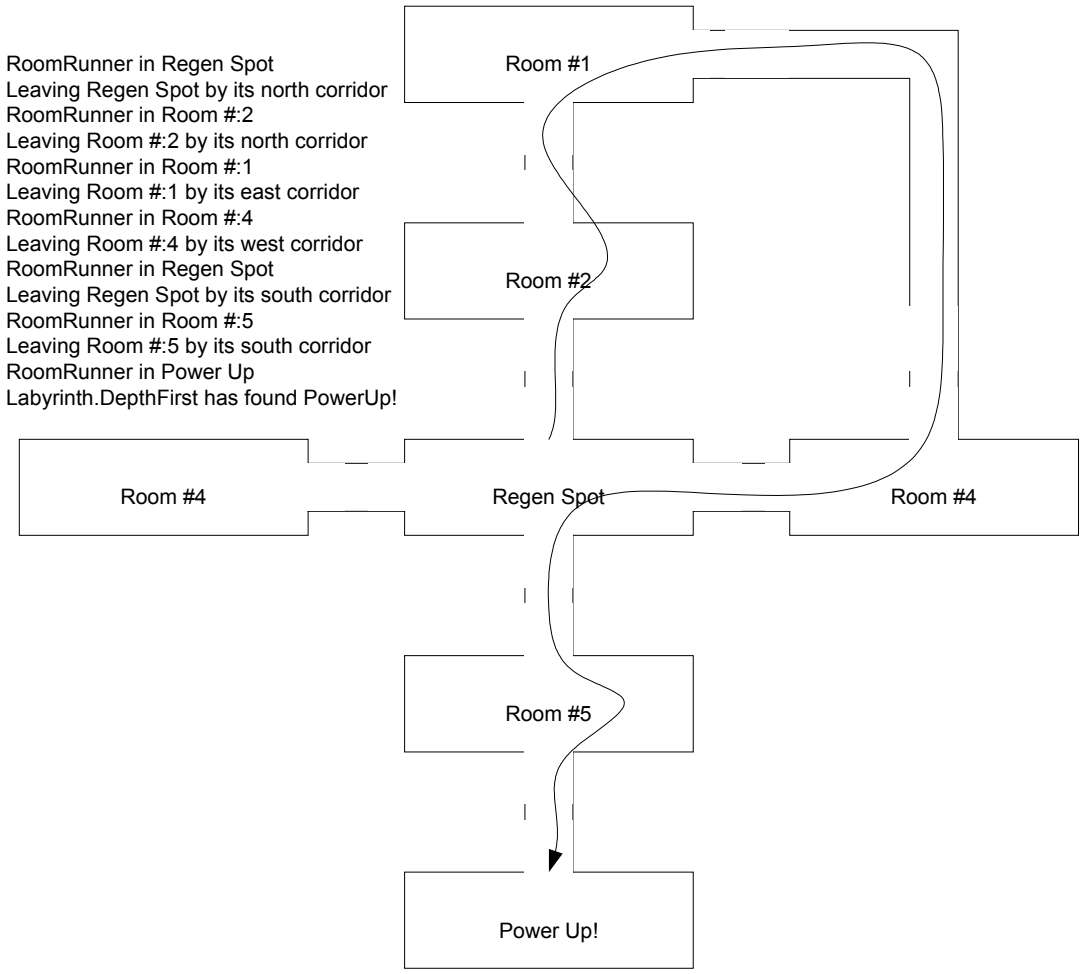

RoomRunner in Regen Spot
Leaving Regen Spot by its north corridor
RoomRunner in Room #:2
Leaving Room #:2 by its north corridor
RoomRunner in Room #:1
Leaving Room #:1 by its east corridor
RoomRunner in Room #:4
Leaving Room #:4 by its west corridor
RoomRunner in Regen Spot
Leaving Regen Spot by its south corridor
RoomRunner in Room #:5
Leaving Room #:5 by its south corridor
RoomRunner in Power Up
Labyrinth.DepthFirst has found PowerUp!
Room #1
Room #2
Room #4
Regen Spot
Room #4
Room #5
Power Up!

Figure 10-6: Custom enumerators allow complex data-structure traversal

Sorting and searching Lists

Utility methods sort and search in **List**s have the same names and signatures as those for arrays of objects, but are instance methods of **IList** instead of **Array**. Here's an example:

```
//:c10:ListSortSearch.cs
// Sorting and searching ILists
using System;
using System.Collections;

public class ListSortSearch {
  private static void Fill(ArrayList list){
    for (int i = 0; i < 25; i++) {
      char c = (char) ('A' + i);
      list.Add(c);
    }
  }

  private static void Print(ArrayList list){
    foreach(Object o in list){
      Console.Write(o + ", ");
    }
    Console.WriteLine();
  }

  private static void Shuffle(ArrayList list){
    int len = list.Count;
    for (int i = 0; i < len; i++) {
      int k = rand.Next(len);
      Object temp = list[i];
      list[i] = list[k];
      list[k] = temp;
    }
  }

  static Random rand = new Random();

  public static void Main() {
    ArrayList list = new ArrayList();
    Fill(list);
```

```
    Print(list);
    Shuffle(list);
    Console.WriteLine("After shuffling: ");
    Print(list);
    list.Reverse();
    Console.WriteLine("Reversed: ");
    Print(list);
    list.Sort();
    Console.WriteLine("After sorting: ");
    Print(list);
    Object key = list[12];
    int index = list.BinarySearch(key);
    Console.WriteLine(
      "Location of {0} is {1}, list[{2}] = {3}",
      key, index, index, list[index]);
  }
} ///:~
```

The use of these methods is identical to the **static** ones in **Array**, but are instance methods of **ArrayList**. This program also contains an implementation of Donald Knuth's shuffling algorithm to randomize the order of a **List**.

From collections to arrays

The **ICollection** interface specifies that all collections must be able to copy their contents into an array. The destination array must be a one-dimensional array with zero-based indexing. The copying procedure may insert objects into the array starting at an arbitrary index, assuming of course that the index is zero or positive and that the destination array is properly initialized.

Copying an **ArrayList** to an array is simple:

```
//:c10:ListToArray.cs
using System;
using System.Collections;

class ListToArray {
  public static void Main(){
    IList list = new ArrayList();
    Random r = new Random();
    int iToAdd = 10 + r.Next(10);
    for (int i = 0; i < iToAdd; i++) {
      list.Add(r.Next(100));
```

```
    }
    int indexForCopyStart = r.Next(10);
    int[] array = new int[
      indexForCopyStart + list.Count];
    list.CopyTo(array, indexForCopyStart);
    for (int i = 0; i < array.Length; i++) {
      Console.WriteLine(
        "array[{0}] = {1}", i, array[i]);
    }
  }
}///:~
```

After initializing an **ArrayList**, we use a random number generator to choose to add between 10 and 19 items. We loop, using **list.Add( )** to add random numbers between 0 and 99. Then, we choose a random number to indicate where in the array we wish to begin copying. We then declare and initialize the array, which must be sized to accommodate **indexForCopyStart** empty integers (since it's an array of **int**s, these will be initialized to 0) and **list.Count** integers from the **ArrayList**. The **CopyTo( )** method takes two parameters – the reference to the destination array and the starting index for the copy. We then loop over the array, outputting the contents.

Since integers are value types, modifying values in the destination **array** will not be reflected in the **ArrayList list**. However, reference types would naturally copy the reference, not the value, and with a reference type, modifying a value referenced in the destination **array** *will* modify the value reference in the **ArrayList**. (To use our metaphor from Chapter 2, you've created new TV remote controls, not new TVs). A copy operation that just copies references is often referred to as a "shallow copy." A "deep copy," in contrast, is one where the referenced objects and all their associated objects are cloned (new TVs, new viewers, new couches, as it were).

Since **IDictionary** inherits from **ICollection**, implementing types must support **CopyTo( )**. The results are an array of **DictionaryEntry** items:

```
//:c10:DictionaryToArray.cs
using System;
using System.Collections;

class DictionaryToArray {
  public static void Main(){
    IDictionary dict = new Hashtable();
    Random r = new Random();
    int iKeys = 3 + r.Next(3);
```

```
    for (int i = 0; i < iKeys; i++) {
      dict.Add(i, r.Next(100));
    }
    DictionaryEntry[] a =
    new DictionaryEntry[dict.Count];
    dict.CopyTo(a, 0);
    for (int i = 0; i < a.Length; i++) {
      DictionaryEntry de = a[i];
      Console.WriteLine(
        "a[{0}]: .Key = {1} .Value = {2}",
        i, de.Key, de.Value);
    }
  }
}///:~
```

A typical run looks like:

```
a[0]: .Key = 4 .Value = 11
a[1]: .Key = 3 .Value = 5
a[2]: .Key = 2 .Value = 6
a[3]: .Key = 1 .Value = 93
a[4]: .Key = 0 .Value = 4
```

You'll note that the resulting array is not sorted on the value of the key, which might be desirable, because **IDictionary** doesn't require keys to be **IComparable**. However, they often are and, if so, it would probably be nice if the resulting array were ordered by key. This program demonstrates a technique to get this result:

```
//:c10:DictionaryToSortedArray.cs
using System;
using System.Collections;

class DictionaryToArray {
  public static void Main(){
    IDictionary dict = new Hashtable();
    Random r = new Random();
    int iKeys = 3 + r.Next(3);
    for (int i = 0; i < iKeys; i++) {
      dict.Add(i, i);
    }

    //First, get array of keys
```

```
    ICollection keyCol = dict.Keys;
    IComparable[] keyArray =
    new IComparable[keyCol.Count];
    //Would throw exception if keys not IComparable
    keyCol.CopyTo(keyArray, 0);

    //Second, get array of values
    ICollection valCol = dict.Values;
    Object[] valArray = new Object[valCol.Count];
    valCol.CopyTo(valArray, 0);

    Array.Sort(keyArray, valArray);

    //Instantiate destination array
    DictionaryEntry[] a =
    new DictionaryEntry[keyCol.Count];
    //Retrieve and set in key-sorted order
    for (int i = 0; i < a.Length; i++) {
      a[i] = new DictionaryEntry();
      a[i].Key = keyArray[i];
      a[i].Value = valArray[i];
    }

    //Output results
    for (int i = 0; i < a.Length; i++) {
      DictionaryEntry de = a[i];
      Console.WriteLine(
        "a[{0}]: .Key = {1} .Value = {2}",
        i, de.Key, de.Value);
    }
  }
}///:~
```

The program starts off similarly to the previous examples, with a few key-value pairs being inserted into a **Hashtable**. Instead of directly copying to the destination array, though, we retrieve the **ICollection** of keys. **ICollection** doesn't have any sorting capabilities, so we use **CopyTo( )** to move the keys into an **IComparable[]** array. If any of our keys did *not* implement **IComparable**, this would throw an **InvalidCastException**.

The next step is to copy the values from the **Hashtable** into another array, this time of type **object[]**. We then use **Array.Sort(Array, Array)**, which sorts *both*

its input arrays based on the comparisons in the first array, which in our case is the key array. In general, one should avoid a situation where one changes the state of one object (such as the array of values) based on logic internal to another object (such as the sorting of the array of keys), a situation that's called *logical association* (see Chapter 9). We could avoid using **Array.Sort(Array, Array)** by sorting **keyArray** and then using a **foreach(object key in keyArray)** loop to retrieve the values from the Hashtable, but in this case that's closing the barn door after the horses have fled – the .NET Framework does not have an **IDictionary** which maintains its objects in key order, which would be the best solution for the general desire to move between an **IDictionary** and a sorted array.

In keeping with its singular nature, **NameValueCollection.CopyTo( )** does not act like **Hashtable.CopyTo( )**. Where **Hashtable.CopyTo( )** creates an array of **DictionaryEntry** objects that contain both the key and value, **NameValueCollection**'s **CopyTo( )** method creates an array of **string**s which represent the values in a comma-separated format, with no reference to the keys. It's difficult to imagine the utility of this format.

This example builds a more reasonable array from a NameValueCollection:

```
//:c10:NValColToArray.cs
using System;
using System.Collections;
using System.Collections.Specialized;

class NameValueCollectionEntry : IComparable {
  string key;
  public string Key{
    get { return key;}
  }

  string[] values;
  public string[] Values{
    get { return values;}
  }

  public NameValueCollectionEntry(
    string key, string[] values){
    this.key = key;
    this.values = values;
  }
```

```
  public int CompareTo(Object o){
    if (o is NameValueCollectionEntry) {
      NameValueCollectionEntry that =
        (NameValueCollectionEntry) o;
      return this.Key.CompareTo(that.Key);
    }
    throw new ArgumentException();
  }

  public static NameValueCollectionEntry[]
    FromNameValueCollection(NameValueCollection src){
    string[] keys = src.AllKeys;
    NameValueCollectionEntry[] results =
    new NameValueCollectionEntry[keys.Length];
    for (int i = 0; i < keys.Length; i++) {
      string key = keys[i];
      string[] vals = src.GetValues(key);
      NameValueCollectionEntry entry =
      new NameValueCollectionEntry(key, vals);
      results[i] = entry;
    }
    return results;
  }
}

class NValColToArray {
  public static void Main(){
    NameValueCollection dict =
      new NameValueCollection();
    Random r = new Random();
    int iKeys = 3 + r.Next(3);
    for (int i = 0; i < iKeys; i++) {
      int iValsToAdd = 5 + r.Next(5);
      for (int j = 0; j < iValsToAdd; j++) {
        dict.Add(i.ToString(),
          r.Next(100).ToString());
      }
    }

    NameValueCollectionEntry[] a =
      NameValueCollectionEntry.
```

```
      FromNameValueCollection(dict);

    Array.Sort(a);
    for (int i = 0; i < a.Length; i++) {
      Console.WriteLine(
        "a[{0}].Key = {1}", i, a[i].Key);

      foreach(string v in a[i].Values){
        Console.WriteLine(
          "\t Value: " + v);
      }
    }
  }
}///:~
```

We define a new type, **NameValueCollectionEntry**, which corresponds to the **DictionaryEntry** of **Hashtable**. Like that type, our new type has a **Key** and a **Value** property, but these are of type **string** and **string[]** respectively. Because we know that the **Key** is always going to be a **string**, we can declare **NameValueCollectionEntry** to implement **IComparable** and implement that simply by comparing **Key** properties (if the parameter to **CompareTo( )** is not a **NameValueCollectionEntry**, we throw an **ArgumentException**).

The static **FromNameValueCollection( )** method is where we convert a **NameValueCollection** into an array of **NameValueCollectionEntry**s. First, we get a **string[]** of keys from the **AllKeys** property of the input parameter (if we had used the **Keys** property, we would have received the same data, but in an **object** array). The **Length** propery of the **keys** allows us to size the **results** array. The **GetValues( )** method of **NameValueCollection** returns a **string** array, which along with the **string** key, is what we need to instantiate a single **NameValueCollectionEntry**. This entry is added to the **results** which are returned when the loop ends.

Class **NValColToArray** demonstrates the use of this new class we've written. A **NameValueCollection** is created and each entry is filled with a random number of strings. A **NameValueCollectionEntry** array called **a** is generated using the static function just discussed. Since we implemented **IComparable** in **NameValueCollectionEntry**, we can use **Array.Sort( )** to sort the results by the **Key string**s. For each **NameValueCollectionEntry**, we output the **Key,** retrieve the **string[] Values**, and output them to the console. We could, of course, sort the **Values** as well, if that was desired.

Persistent data with ADO.NET

While collection classes and data structures remain important to in-memory manipulation of data, offline storage is dominated by third-party vendors supporting the relational model of data storage. Of course, Oracle's eponymous product dominates the high-end market, while Microsoft's SQL Server and IBM's DB2 are able competitors for enterprise data. There are hundreds of databases appropriate for smaller projects, the most well-distributed of which is Microsoft's Access.

Meanwhile, the success of the Web made many people comfortable with the concept that "just" text-based streams marked with human-readable tags were sufficiently powerful to create a lot of end-user value. Extensible Markup Language (XML) has exploded in popularity in the new millennium and is rapidly becoming the preferred in-memory representation of relational data. This will be discussed in depth in Chapter 17, but some discussion of XML is relevant to any discussion of Active Data Objects for .NET (ADO.NET).

Like graphics programming, the complete gamut of database programming details cannot be adequately covered in less than an entire book. However, also like graphics programming, most programmers do not need to know more than the basics. In the case of database programming, 99% of the work boils down to being able to Create, Read, Update, and Delete data – the functions known as "CRUD." This section tries to deliver the *minimum* amount of information you need to be able to use ADO.NET in a professional setting.

Although CRUD may encapsulate many programmers intent for ADO.NET, there's another "D" that is fundamental to ADO.NET – "Disconnected." Ironically, the more the World Wide Web becomes truly ubiquitous, the more difficult it is to create solutions based on continuous connections. The client software that is running in a widely distributed application, i.e., an application that is running over the Internet, simply cannot be counted on to go through an orderly, timely lifecycle of "connect, gain exclusive access to data, conduct transactions, and disconnect." Similarly, although continuous network access may be the rule in corporate settings, the coming years are going to see an explosion in applications for mobile devices, such as handhelds and telephones, which have metered costs; economics dictate that such applications cannot be constantly connected.

ADO.NET separates the tasks of actually accessing the data store from the tasks of manipulating the resulting set of data. The former obviously require a connection to the store while the latter do not. This separation of concerns helps when converting data from one data store to another (such as converting data between

relational table data and XML) as well as making it much easier to program widely-distributed database applications. However, this model increases the possibility that two users will make incompatible modifications to related data – they'll both reserve the last seat on the flight, one will mark an issue as resolved while the other will expand the scope of the investigation, etc. So even a minimal introduction to ADO.NET requires some discussion of the issues of concurrency violations.

Getting a handle on data with DataSet

The **DataSet** class is the root of a relational view of data. A **DataSet** has **DataTable**s, which have **DataColumn**s that define the types in **DataRow**s. The relational database model was introduced by Edgar F. Codd in the early 1970s. The concept of tables storing data in rows in strongly-typed columns may seem to be the very definition of what a database is, but Codd's formalization of these concepts and others such such as *normalization* (a process by which redundant data is eliminated and thereby ensuring the correctness and consistency of edits) was one of the great landmarks in the history of computer science.

While normally one creates a **DataSet** based on existing data, it's possible to create one from scratch, as this example shows:

```
//:c10:BasicDataSetOperations.cs
using System;
using System.Data;

class BasicDataSetOperations {
  public static void Main(string[] args){
    DataSet ds = BuildDataSet();
    PrintDataSetCharacteristics(ds);
  }

  private static DataSet BuildDataSet() {
    DataSet ds = new DataSet("MockDataSet");

    DataTable auTable = new DataTable("Authors");
    ds.Tables.Add(auTable);

    DataColumn nameCol = new DataColumn("Name",
      typeof(string));
    auTable.Columns.Add(nameCol);

    DataRow larryRow = auTable.NewRow();
```

```
    larryRow["Name"] = "Larry";
    auTable.Rows.Add(larryRow);
    DataRow bruceRow = auTable.NewRow();
    bruceRow["Name"] = "Bruce";
    auTable.Rows.Add(bruceRow);

    return ds;
  }

  private static void PrintDataSetCharacteristics(
    DataSet ds){
    Console.WriteLine(
      "DataSet \"{0}\" has {1} tables",
      ds.DataSetName, ds.Tables.Count);
    foreach(DataTable table in ds.Tables){
      Console.WriteLine(
        "Table \"{0}\" has {1} columns",
        table.TableName, table.Columns.Count);
      foreach(DataColumn col in table.Columns){
        Console.WriteLine(
          "Column \"{0}\" contains data of type {1}",
          col.ColumnName, col.DataType);
      }

      Console.WriteLine(
        "The table contains {0} rows",
        table.Rows.Count);
      foreach(DataRow r in table.Rows){
        Console.Write("Row Data: ");
        foreach(DataColumn col in table.Columns){
          string colName = col.ColumnName;
          Console.Write("[{0}] = {1}",
            colName, r[colName]);
        }
        Console.WriteLine();
      }
    }
  }
}///:~
```

The .NET classes related to **DataSets** are in the **System.Data** namespace, so naturally we have to include a **using** statement at the beginning of the program.

The **Main( )** method is straightforward: it calls **BuildDataSet( )** and passes the object returned by that method to another static method called **PrintDataSetCharacteristics( )**.

BuildDataSet( ) introduces several new classes. First comes a **DataSet**, using a constructor that allows us to simultaneously name it "MockDataSet." Then, we declare and initialize a **DataTable** called "Author" which we reference with the **auTable** variable. **DataSet** objects have a **Tables** property of type **DataTableCollection**, which implements **ICollection**. While **DataTableCollection** does not implement **IList**, it contains some similar methods, including **Add**, which is used here to add the newly created **auTable** to **ds**'s **Tables**.

DataColumns, such as the **nameCol** instantiated in the next line, are associated with a particular **DataType**. **DataType**s are not nearly as extensive or extensible as normal types. Only the following can be specified as a **DataType**:

Boolean	DateTime
Decimal	Double
Int16	Int32
Int64	SByte
Single	String
TimeSpan	UInt16
UInt32	UInt64

In this case, we specify that the "Name" column should store strings. We add the column to the **Columns** collection (a **DataColumnCollection**) of our **auTable**.

One cannot create rows of data using a standard constructor, as a row's structure must correspond to the **Columns** collection of a particular **DataTable**. Instead, **DataRow**s are constructed by using the **NewRow( )** method of a particular **DataTable**. Here, **auTable.NewRow( )** returns a **DataRow** appropriate to our "Author" table, with its single "Name" column. **DataRow** does not implement **ICollection**, but does overload the indexing operator, so assigning a value to a column is as simple as saying: `larryRow["Name"] = "Larry".`

The reference returned by **NewRow( )** is *not* automatically inserted into the **DataTable** which generates it; that is done by:

```
        auTable.Rows.Add(larryRow);
```

After creating another row to contain Bruce's name, the **DataSet** is returned to the **Main( )** method, which promptly passes it to **PrintDataSetCharacteristics( )**. The output is:

```
DataSet "MockDataSet" has 1 tables
Table "Authors" has 1 columns
Column "Name" contains data of type System.String
The table contains 2 rows
Row Data: [Name] = Larry
Row Data: [Name] = Bruce
```

Connecting to a database

The task of actually moving data in and out of a store (either a local file or a database server on the network) is the task of the **IDbConnection** interface. Specifying which data (from all the tables in the underlying database) is the responsibility of objects which implement IDbCommand. And bridging the gap between these concerns and the concerns of the **DataSet** is the responsibility of the **IDbAdapter** interface.

Thus, while **DataSet** and the classes discussed in the previous example encapsulate the "what" of the relational data, the **IDataAdapter**, **IDbCommand**, and **IDbConnection** encapsulate the "How":

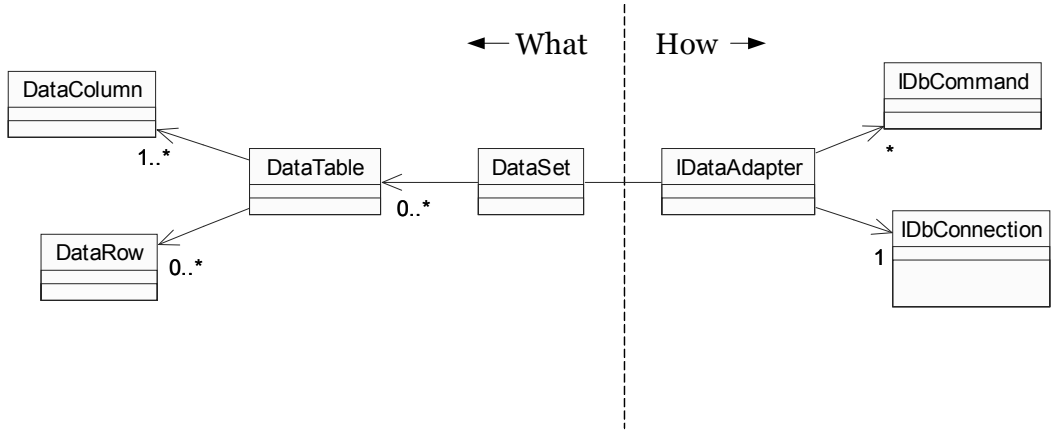


Figure 10-7: ADO.NET separates the "What data" classes from the "How we get it" classes

The .NET Framework currently ships with two *managed providers* that implement **IDataAdapter** and its related classes. One is high-performance provider

optimized for Microsoft SQL Server; it is located in the System.Data.SqlClient namespace. The other provider, in the System.Data.OleDb namespace, is based on the broadly available Microsoft JET engine (which ships as part of Windows XP and is downloadable from Microsoft's Website). Additionally, you can download an ODBC-suppporting managed provider from msdn.microsoft.com. One suspects that high-performance managed providers for Oracle, DB2, and other high-end databases will quietly become available as .NET begins to achieve significant market share.

For the samples in this chapter, we're going to use the OleDb classes to read and write an Access database, but we're going to upcast everything to the ADO.NET interfaces so that the code is as general as possible.

The "Northwind" database is a sample database from Microsoft that you can download from *http://msdn.microsoft.com/downloads* if you don't already have it on your hard-drive from installing Microsoft Access. The file is called "nwind.mdb". Unlike with enterprise databases, there is no need to run a database server to connect to and manipulate an Access database. Once you have the file you can begin manipulating it with .NET code.

This first example shows the basic steps of connecting to a database and filling a dataset:

```
//:c10:DBConnect.cs
using System;
using System.Data;
using System.Data.OleDb;

class BasicDataSetOperations {
  public static void Main(string[] args){
    DataSet ds = Employees("Nwind.mdb");
    Console.WriteLine(
      "DS filled with {0} rows",
      ds.Tables[0].Rows.Count);
  }
  private static DataSet Employees(string fileName){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString=
    "Provider=Microsoft.JET.OLEDB.4.0;" +
    "data source=" + fileName;
    DataSet ds = null;
    try {
```

```
      cnctn.Open();

      string selStr = "SELECT * FROM EMPLOYEES";
      IDataAdapter adapter =
      new OleDbDataAdapter(selStr, cnctn);

      ds = new DataSet("Employees");
      adapter.Fill(ds);
    } finally {
      cnctn.Close();
    }

    return ds;
  }
}///:~
```

After specifying that we'll be using the System.Data and System.Data.OleDb namespaces, the **Main( )** initializes a **DataSet** with the results of a call to the static function **Employees( )**. The number of rows in the first table of the result is printed to the console.

The method **Employees( )** takes a string as its parameter in order to clarify the part of the connection string that is variable. In this case, you'll obviously have to make sure that the file "Nwind.mdb" is in the current directory or modify the call appropriately.

The **ConnectionString** property is set to a bare minimum: the name of the provider we intend to use and the data source. This is all we need to connect to the Northwind database, but enterprise databases will often have significantly more complex connection strings.

The call to **cnctn.Open( )** starts the actual process of connecting to the database, which in this case is a local file read but which would typically be over the network. Because database connections are the prototypical "valuable non-memory resource," as discussed in Chapter 11, we put the code that interacts with the database inside a **try...finally** block.

As we said, the **IDataAdapter** is the bridge between the "how" of connecting to a database and the "what" of a particular relational view into that data. The bridge going from the database to the **DataSet** is the **Fill( )** method (while the bridge from the **DataSet** to the database is the **Update( )** method, which we'll discuss in our next example). How does the **IDataAdapter** know what data to put into the **DataSet**? The answer is actually not defined at the level of **IDataAdapter**. The

OleDbAdapter supports several possibilities, including automatically filling the **DataSet** with all, or a specified subset, of records in a given table. The **DBConnect** example shows the use of Structured Query Language (SQL), which is probably the most general solution. In this case, the SQL query `SELECT * FROM EMPLOYEES` retrieves all the columns and all the data in the EMPLOYEES table of the database.

The **OleDbDataAdapter** has a constructor which accepts a string (which it interprets as a SQL query) and an **IDbConnection**. This is the constructor we use and upcast the result to **IDataAdapter**.

Now that we have our open connection to the database and an **IDataAdapter,** we create a new **DataSet** with the name "Employees." This empty **DataSet** is passed in to the **IDataAdapter.Fill( )** method, which executes the query via the **IDbConnection**, adds to the passed-in **DataSet** the appropriate **DataTable** and **DataColumn** objects that represent the structure of the response, and then creates and adds to the **DataSet** the **DataRow** objects that represent the results.

The **IDbConnection** is **Close**d within a **finally** block, just in case an **Exception** was thrown sometime during the database operation. Finally, the filled **DataSet** is returned to **Main( )**, which dutifully reports the number of employees in the Northwind database.

Fast reading with IDataReader

The preferred method to get data is to use an **IDataAdapter** to specify a view into the database and use **IDataAdapter.Fill( )** to fill up a **DataSet.** An alternative, if all you want is a read-only forward read, is to use an **IDataReader**. An **IDataReader** is a direct, connected iterator of the underlying database; it's likely to be more efficient than filling a **DataSet** with an **IDataAdapter**, but the efficiency requires you to forego the benefits of a disconnected architecture. This example shows the use of an **IDataReader** on the **Employees** table of the Northwind database:

```
//:c10:DataReader.cs
using System;
using System.Data;
using System.Data.OleDb;

class DataReader {
  public static void Main(){
    EnumerateEmployees("Nwind.mdb");
  }
```

```
  private static void EnumerateEmployees(string fileName){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString=
    "Provider=Microsoft.JET.OLEDB.4.0;" +
    "data source=" + fileName;
    IDataReader rdr = null;
    try {
      cnctn.Open();
      IDbCommand sel =
      new OleDbCommand("SELECT * FROM EMPLOYEES", cnctn);
      rdr = sel.ExecuteReader();
      while (rdr.Read()) {
        Console.WriteLine(rdr["FirstName"] + " "
          + rdr["LastName"]);
      }
    } finally {
      rdr.Close();
      cnctn.Close();
    }
  }
}///:~
```

The **EnumerateEmployees( )** method starts like the code in the **DBConnect** example, but we do not upcast the **OleDbConnection** to **IDbConnection** for reasons we'll discuss shortly. The connection to the database is identical, but we declare an **IDataReader rdr** and initialize it to null before opening the database connection; this is so that we can use the **finally** block to **Close( )** the **IDataReader** as well as the **OleDbConnection**.

After opening the connection to the database, we create an **OleDbCommand** which we upcast to **IDbCommand**. In the case of the **OleDbCommand** constructor we use, the parameters are a SQL statement and an **OleDbConnection** (thus, our inability to upcast in the first line of the method).

The next line, **rdr = sel.ExecuteReader( )**, executes the command and returns a *connected* **IDataReader**. **IDataReader.Read( )** reads the next line of the query's result, returning **false** when it runs out of rows. Once all the data is read, the method enters a **finally** block, which severs the **IDataReader**'s connection with **rdr.Close( )** and then closes the database connection entirely with **cnctn.Close( )**.

CRUD with ADO.NET

With **DataSet**s and managed providers in hand, being able to create, read, update, and delete records in ADO.NET is near at hand. Creating data was covered in the **BasicDataSetOperations** example – use **DataTable.NewRow( )** to generate an appropriate **DataRow**, fill it with your data, and use **DataTable.Rows.Add( )** to insert it into the **DataSet**. Reading data is done in a flexible disconnected way with an **IDataAdapter** or in a fast but connected manner with an **IDataReader**.

Update and delete

The world would be a much pleasanter place if data never needed to be changed or erased[2]. These two operations, especially in a disconnected mode, raise the distinct possibility that two processes will attempt to perform incompatible manipulation of the same data. There are two options for a database model:

- Assume that any read that might end in an edit *will* end in an edit, and therefore not allow anyone else to do a similar editable read. This model is known as pessimistic concurrency.
- Assume that although people will edit and delete rows, make the enforcement of consistency the responsibility of some software component other than the database components. This is optimistic concurrency, the model that ADO.NET uses.

When an **IDbAdapter** attempts to update a row that has been updated since the row was read, the second update fails and the adapter throws a **DBConcurrencyException** (note the capital 'B' that violates .NET's the naming convention).

As an example:

1. Ann and Ben both read the database of seats left on the 7 AM flight to Honolulu. There are 7 seats left.
2. Ann and Ben both select the flight, and their client software shows 6 seats left.
3. Ann submits the change to the database and it completes fine.
4. Charlie reads the database, sees 6 seats available on the flight.

---

[2] Not only would it please the hard drive manufacturers, it would provide a way around the second law of thermodynamics. See, for instance, *http://www.media.mit.edu/physics/publications/papers/96.isj.ent.pdf*

5. Ben submits the change to the database. Because Ann's update happened before Ben's update, Ben receives a **DBConcurrencyException**. The database does *not* accept Ben's change.

6. Charlie selects a flight and submits the change. Because the row hasn't changed since Charlie read the data, Charlie's request succeeds.

It is impossible to give even general advice as to what to do after receiving a **DBConcurrencyException**. Sometimes you'll want to take the data and re-insert it into the database as a new record, sometimes you'll discard the changes, and sometimes you'll read the new data and reconcile it with your changes. There are even times when such an exception indicates a deep logical flaw that calls for a system shutdown.

This example performs all of the CRUD operations, rereading the database after the update so that the subsequent deletion of the new record does not throw a **DBConcurrencyException**:

```
//:c10:Crud.cs
using System;
using System.Data;
using System.Data.OleDb;

class Crud {
  public static void Main(string[] args){
    Crud myCrud = new Crud();
    myCrud.ReadEmployees("NWind.mdb");
    myCrud.Create();
    myCrud.Update();
    //Necessary to avoid DBConcurrencyException
    myCrud.Reread();
    myCrud.Delete();
  }

  OleDbDataAdapter adapter;
  DataSet emps;

  private void ReadEmployees(string pathToAccessDB){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString =
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + pathToAccessDB;
```

```
    cnctn.Open();

    string selStr = "SELECT * FROM EMPLOYEES";
    adapter = new OleDbDataAdapter(selStr, cnctn);
    new OleDbCommandBuilder(adapter);

    emps = new DataSet("Employees");
    adapter.Fill(emps);
  }

  private void Create(){
    DataRow r = emps.Tables["Table"].NewRow();
    r["FirstName"] = "Bob";
    r["LastName"] = "Dobbs";
    emps.Tables["Table"].Rows.Add(r);
    adapter.Update(emps);
  }

  private void Update(){
    DataRow aRow = emps.Tables["Table"].Rows[0];
    Console.WriteLine("First Name: "
      + aRow["FirstName"]);
    string newName = null;
    if (aRow["FirstName"].Equals("Nancy")) {
      newName = "Adam";
    } else {
      newName = "Nancy";
    }
    aRow.BeginEdit();
    aRow["FirstName"] = newName;
    aRow.EndEdit();
    Console.WriteLine("First Name: "
      + aRow["FirstName"]);
    //Update only happens now
    int iChangedRows = adapter.Update(emps);
    Console.WriteLine("{0} rows updated",
      iChangedRows);
  }

  private void Reread(){
    adapter.Fill(emps);
```

```
  }

  private void Delete(){
    //Seems to return 1 greater than actual count
    int iRow = emps.Tables["Table"].Rows.Count;
    DataRow lastRow =
    emps.Tables["Table"].Rows[iRow - 1];
    Console.WriteLine("Deleting: "
      + lastRow["FirstName"]);
    lastRow.Delete();
    int iChangedRows = adapter.Update(emps);
    Console.WriteLine("{0} rows updated",
      iChangedRows);
  }
}///:~
```

The **Main( )** method outlines what we're going to do: read the "Employees" table, create a new record, update a record, reread the table (you can comment out the call to **Reread( )** if you want to see a **DBConcurrencyException**), and delete the record we created.

The **Crud** class has instance variables for holding the **OleDbDataAdapter** and **DataSet** that the various methods will use. **ReadEmployees( )** opens the database connection and creates the adapter just as we've done before.

The next line:

```
new OleDbCommandBuilder(adapter);
```

demonstrates a utility class that automatically generates and sets within the **OleDbDataAdapter** the SQL statements that insert, update, and delete data in the same table acted on by the select command. **OleDbCommandBuilder** is very convenient for SQL data adapters that work on a single table (there's a corresponding **SqlCommandBuilder** for use with SQL Server). For more complex adapters that involve multiple tables, you have to set the corresponding **InsertCommand**, **DeleteCommand**, and **UpdateCommand** properties of the **OleDbDataAdapter**. These commands are needed to commit to the database changes made in the **DataSet.**

The first four lines of method **Create( )** show operations on the **DataSet emps** that we've seen before – the use of **Table.NewRow( ),** and **DataRowCollection.Add( )** to manipulate the **DataSet**. The final line calls **IDataAdapter.Update( )**, which attempts to commit the changes in the **DataSet**

to the backing store (it is this method which requires the SQL commands generated by the **OleDbCommandBuilder**).

The method **Update( )** begins by reading the first row in the **emps DataSet**. The call to **DataRow.BeginEdit( )** puts the **DataRow** in a "Proposed" state. Changes proposed in a **DataRow** can either be accepted by a call to **DataRow.EndEdit( )** or the **AcceptChanges( )** method of either the **DataRow**, **DataTable**, or **DataSet**. They can be cancelled by a call to **DataRow.CancelEdit( )** or the **RejectChanges( )** methods of the classes.

After printing the value of the first row's "FirstName" column, we put **aRow** in a "Proposed" state and change the "FirstName" to "Fred." We call **CancelEdit( )** and show on the console that "Fred" *is not* the value. If the first name is currently "Nancy" we're going to change it to "Adam" and vice versa. This time, after calling **BeginEdit( )** and making the change, we call **EndEdit( )**. At this point, the data is changed in the **DataSet**, but not yet in the database. The database commit is performed in the next line, with another call to **adapter.Update( )**.

This call to **Update( )** succeeds, as the rows operated on by the two calls to **Update( )** are different. If, however, we were to attempt to update either of these two rows without rereading the data from the database, we would get the dread **DBConcurrencyException**. Since deleting the row we added is exactly our intent, **Main( )** calls **Reread( )**, which in turn calls **adapter.Fill( )** to refill the **emps DataSet**.

Finally, **Main( )** calls **Delete( )**. The number of rows is retrieved from the **Rows** collection. But because the index into rows is 0-based, we need to subtract 1 from the total count to get the index of the last row (e.g., the **DataRow** in a **DataTable** with a **Count** of 1 would be accessed at **Rows[0]**). Once we have the last row in the DataSet (which will be the "Bob Dobbs" record added by the **Create( )** method), a call to **DataRow.Delete( )** removes it from the **DataSet** and **DataAdapter.Update( )** commits it to the database.

The object-relational impedance mismatch

If you ever find yourself unwelcome in a crowd of suspicious programmers, say "I was wondering: what is your favorite technique for overcoming the object-relational impedance mismatch?" This is like a secret handshake in programmer circles: not only does it announce that you're not just some LISP hacker fresh from Kendall Square, it gives your inquisitors a chance to hold forth on A Matter of Great Import.

You can see the roots of the mismatch even in the basic examples we've shown here. It's taken us several pages just to show how to do the equivalent of **new** and assignment to relational data! Although a table is something like a class, and a row is something like an instance of the class, tables have no concept of binding data and behavior into a coherent whole, nor does the standard relational model have any concept of inheritance. Worse, it's become apparent over the years that there's no single strategy for mapping between objects and tables that is appropriate for all needs.

Thinking in Databases would be a very different book than *Thinking in C#*. The object and relational models are very different, but contain just enough similarities so that the pain hasn't been enough to trigger a wholesale movement towards object databases (which have been the Next Big Thing in Programming for more than a decade).

High-performing, highly-reliable object databases are available today, but have no mindshare in the enterprise market. What has gained mindshare is a hybrid model, which combines the repetitive structure of tables and rows with a hierarchical containment model that is closer to the object model. This hybrid model, embodied in XML, does not directly support the more complicated concepts of relational joins or object inheritance, but is a good waypoint on the road to object databases. We'll discuss XML in more detail in Chapter 17 and revisit ADO.NET in our discussion of data-bound controls in Chapter 14.

Summary

To review the tools in the .NET Framework that collect objects:

An array associates numerical indices to objects. It holds objects of a known type so that you don't have to cast the result when you're looking up an object. It can be multidimensional in two ways – rectangular or jagged. However, its size cannot be changed once you create it.

An **IList** holds single elements, an **IDictionary** holds key-value pairs, and a **NameObjectCollectionBase** holds string-Collection pairs.

Like an array, an **IList** also associates numerical indices to objects—you can think of arrays and **IList**s as ordered containers. An **IDictionary** overloads the bracket operator of the array to make it easy to access values, but the underlying implementation is not necessarily ordered.

Most collections automatically resize themselves as you add elements, but the **BitArray** needs to be explicitly sized.

ICollections hold only **object** references, so primitives are boxed and unboxed when stored. With the exception of type-specific containers in System.Collections.Specialized and those you roll yourself, you must always cast the result when you pull an **object** reference out of a container. Type-specific container classes will be supported natively by the .NET run-time sometime in the future.

Data structures have inherent characteristics distinct from the data that is stored in them. Sorting, searching, and traversal have traditionally been matters of great day-to-day import. Advances in abstraction and computer power allow most programmers to ignore most of these issues most of the time, but occasionally produce the most challenging and rewarding opportunities in programming.

ADO.NET provides an abstraction of the relational database model. **DataSet**s represent relational data in memory, while **IDataAdapter**s and related classes move the data in and out of databases.

The collection classes are tools that you use on a day-to-day basis to make your programs simpler, more powerful, and more effective. Thoroughly understanding them and extending and combining them to rapidly solve solutions is one mark of software professionalism.

Exercises

1. Create a new class called **Gerbil** with an **int gerbilNumber** that's initialized in the constructor. Give it a method called **Hop( )** that prints out its **gerbilNumber** and that it's hopping. Create an **ArrayList** and add a bunch of **Gerbil** objects to it. Now use the indexing operator [ ] to move through the **ArrayList** and call **Hop( )** for each **Gerbil**.

2. Modify the previous exercise so you use an **IEnumerator** to move through the **ArrayList** while calling **Hop( )**.

3. Take the **Gerbil** class in and put it into a **Hashtable** instead, associating the name of the **Gerbil** as a **string** (the key) for each **Gerbil** (the value) you put in the table. Get an **IEnumerator** for the **Hashtable.Keys( )** and use it to move through the **Hashtable**, looking up the **Gerbil** for each key and printing out the key and telling the **Gerbil** to **Hop( )**.

4. Create a container that encapsulates an array of string, and that only adds strings and gets strings, so that there are no casting issues during use. If the internal array isn't big enough for the next add, your container should

automatically resize it. In **Main( )**, compare the performance of your container with an **ArrayList** holding **string**s.

5. Create a class containing two **string** objects, and make it comparable so that the comparison only evaluates the first **string**. Fill an array and an **ArrayList** with objects of your class. Demonstrate that sorting works properly.

6. Modify the previous exercise so that an alphabetic sort is used.

7. Create a custom indexer for maze running that implements breadth-first traversal. For every non-visited tunnel out of a room, go to the next room. If it's the end, stop traversing. If it's not the end, return to the original room and try the next option. If none of the rooms out of the original room are the final room, investigate the rooms that are two corridors distant from the original room.

8. Modify the maze-running challenge so that each tunnel traversed has a weight varying from 0 to 1. Use your depth- and bread-first traversals to discover the cheapest route from the beginning to the end.

9. (Challenging) Write a maze-generating program that makes mazes consisting of hundreds or thousands of rooms and tunnels. Find an efficient way to determine the minimum traversal cost. If you can't come up with an efficient way to solve it, prove that there is no efficient way[3].

10. Write a program to read and write to tables in the Northwind database other than Employees.

11. Write a program to CRUD data stored in a SQL Server database.

12. (Challenging) Investigate applications of wavelets in domains such as compression, database retrieval, and signal processing. Develop efficient tools for investigating wavelet applications[4].

---

[3] If you complete this exercise, you will have proved whether or not P ≠ NP. In addition to being a shoe-in for the Turing Award and probably the Fields Medal, you will be eligible for a $1,000,000 prize from the Clay Foundation (*http://www.claymath.org*).

[4] While not as challenging as proving P ≠ NP, there are loads of practical applications for wavelets that are just begging to be written in fields as diverse as video processing, bioinformatics, and Web retrieval.

11: Error Handling with Exceptions

Every program is based on a vast array of expectations. Some expectations are so basic that it doesn't make sense to worry about them – does the current computer have a hard-drive, sufficient RAM to load the program, and so forth. Other expectations are explicit in the code and violations can be discovered at compile-time – this method takes an integer as a parameter, not a string, that method returns a Fish not a Fowl. The majority of expectations, though, are implicit contracts between methods and the client code that calls them. When the reality at runtime is such that an expectation goes unfulfilled, C# uses Exceptions to signal the disruption of the program's expected behavior.

When an object can recognize a problem but does not have the context to intelligently deal with the problem, recovery may be possible. For instance, when a network message is not acknowledged, perhaps a retry is in order, but that decision shouldn't be made at the lowest level (network games, for instance, often have data of varying importance, some of which must be acknowledged and some which would be worthless by the time a retry could be made). On the other hand, a method may have a problem because something is awry with the way it is being used – perhaps a passed-in parameter has an invalid value (a **PrintCalendar** method is called for the month "Eleventember") or perhaps the method can only be meaningfully called when the object is in a different state (for instance, a **Cancel** method is called when an **Itinerary** object is not in a "booked" state).

These misuse situations are tricky because there is no way in C# to specify a method's preconditions and postconditions as an explicit contract – a way in source code to say "if you call me with x, y, and z satisfied, I will guarantee that when I return condition a, b, and c will be satisfied (assuming of course that all

the methods I call fulfill their contractual obligations with me)." For instance, .NET's Math class has a square root function that takes a double as its parameter. Since .NET does not have a class to represent imaginary numbers, this function can only return a meaningful answer if passed a positive value. If this method is called with a negative value, is that an exceptional condition or a disappointing, but predictable, situation? There's no way to tell from the method's signature:

```
double Math.Sqrt(double d);
```

Although preconditions and postconditions are not explicit in C# code, you should always think in terms of contracts while programming and document pre- and postconditions in your method's param and returns XML documentation. The .NET library writers followed this advice and the documentation for **Math.Sqrt( )** explain that it will return a NaN (Not A Number) value if passed a negative parameter.

There is no hard-and-fast rule to determine what is an exceptional condition and what is reasonably foreseeable. Returning a special "invalid value" such as does **Math.Sqrt( )** is debatable, especially if the precondition is not as obvious as "square roots can only be taken on positive numbers."

When an exceptional condition occurs such that a method cannot fulfill its postconditions, there are only two valid things to do: attempt to change the conditions that led to the problem and retry the method, or "organized panic" – put objects into consistent states, close or release non-memory resources, and move control to a much different context that can either perform a recovery or log as much information as possible about the condition leading to the failure to help in debugging efforts. Some people emphasize recovery far too early; until late in the development of a high-availability system it's better to have your system break and trigger a defect-fixing coding session than to cleanup-and-recover and allow the defect to continue.

Both of these valid choices (retrying or cleanup) usually cannot be fully done at the point where the exceptional condition occurred. With a network error sometimes just waiting a half-second or so and retrying may be appropriate, but usually a retry requires changing options at a higher level of abstraction (for instance, a file-writing related error might be retried after giving the user a chance to choose a different location). Similarly, cleanup leading to either recovery or an orderly shutdown may very well require behavior from all the objects in your system, not just those objects referenced by the class experiencing the problem.

When an exceptional condition occurs, it is up to the troubled method to create an object of a type derived from **Exception**. Such objects can be **thrown** so that control moves, not to the next line of code or into a method call as is normally the case, but rather propagates to blocks of code that are dedicated to the tasks of either recovery or cleanup.

The orderly way in which **Exception**s propagate from the point of trouble has two benefits. First, it makes error-handling code hierarchical, like a chain of command. Perhaps one level of code can go through a sequence of options and retry, but if those fail, can give up and propagate the code to a higher level of abstraction, which may perform a clean shutdown. Second, exceptions clean up error handling code. Instead of checking for a particular rare failure and dealing with it at multiple places in your program, you no longer need to check at the point of the method call (since the exception will propagate right out of the problem area to a block dedicated to catching it). And, you need to handle the problem in only one place, the so-called *exception handler*. This saves you code, and it separates the code that describes what you want to do from the code that is executed when things go awry. In general, reading, writing, and debugging code become much clearer with exceptions than with alternative ways of error handling.

This chapter introduces you to the code you need to write to properly handle exceptions, and the way you can generate your own exceptions if one of your methods gets into trouble.

Basic exceptions

When you throw an exception, several things happen. First, the exception object is created in the same way that any C# object is created: on the heap, with **new**. Then the current path of execution (the one you couldn't continue) is stopped and the reference for the exception object is ejected from the current context. At this point the exception handling mechanism takes over and begins to look for an appropriate place to continue executing the program. This appropriate place is the *exception handler,* whose job is to recover from the problem so the program can either retry the task or cleanup and propagate either the original **Exception** or, better, a higher-abstraction **Exception**.

As a simple example of throwing an exception, consider an object reference called **t** that is passed in as a parameter to your method. Your design contract might require as a precondition that **t** refer to a valid, initialized object. Since C# has no syntax for enforcing preconditions, some other piece of code may pass your method a **null** reference and compile with no problem. This is an easy

precondition violation to discover and there's no special information about the problem that you think would be helpful for its handlers. You can send information about the error into a larger context by creating an object representing the problem and its context and "throwing" it out of your current context. This is called *throwing an exception*. Here's what it looks like:

```
if(t == null)
  throw new ArgumentNullException();
```

This throws the exception, which allows you—in the current context—to abdicate responsibility for thinking about the issue further. It's just magically handled somewhere else. Precisely *where* will be shown shortly.

Exception arguments

Like any object in C#, you always create exceptions on the heap using **new**, which allocates storage and calls a constructor. There are four constructors in all standard exceptions:

- The default, no argument constructor
- A constructor that takes a string as a message:
  throw new ArgumentNullException("t");
- A constructor that takes a message and an inner, lower-level Exception:
  throw new PreconditionViolationException("invalid t", new ArgumentNullException("t"))
- And a constructor specifically designed for Remoting (.NET Remoting is not covered in this book)

The keyword **throw** causes a number of relatively magical things to happen. Typically, you'll first use **new** to create an object that represents the error condition. You give the resulting reference to **throw**. The object is, in effect, "returned" from the method, even though that object type isn't normally what the method is designed to return. A simplistic way to think about exception handling is as an alternate return mechanism, although you get into trouble if you take that analogy too far. You can also exit from ordinary scopes by throwing an exception. But a value is returned, and the method or scope exits.

Any similarity to an ordinary return from a method ends here, because *where* you return is someplace completely different from where you return for a normal method call. (You end up in an appropriate exception handler that might be miles away—many levels away on the call stack—from where the exception was thrown.)

Typically, you'll throw a different class of exception for each different type of error. The information about the error is represented both inside the exception object and implicitly in the type of exception object chosen, so someone in the bigger context can figure out what to do with your exception. (Often, it's fine that the only information is the type of exception object, and nothing meaningful is stored within the exception object.)

Catching an exception

If a method throws an exception, it must assume that exception is "caught" and dealt with. One of the advantages of C#'s exception handling is that it allows you to concentrate on the problem you're trying to solve in one place, and then deal with the errors from that code in another place.

To see how an exception is caught, you must first understand the concept of a *guarded region,* which is a section of code that might produce exceptions, and which is followed by the code to handle those exceptions.

The try block

If you're inside a method and you throw an exception (or another method you call within this method throws an exception), that method will exit in the process of throwing. If you don't want a **throw** to exit the method, you can set up a special block within that method to capture the exception. This is called the *try block* because you "try" your various method calls there. The try block is an ordinary scope, preceded by the keyword **try**:

```
try {
  // Code that might generate exceptions
}
```

If you were checking for errors carefully in a programming language that didn't support exception handling, you'd have to surround every method call with setup and error testing code, even if you call the same method several times. With exception handling, you put everything in a try block and capture all the exceptions in one place. This means your code is a lot easier to write and easier to read because the goal of the code is not confused with the error checking.

Exception handlers

Of course, the thrown exception must end up someplace. This "place" is the *exception handler,* and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3) {
  // Handle exceptions of Type3 without needing ref
}

// etc...
```

Each catch clause (exception handler) is like a little method that takes one and only one argument of a particular type. The identifier (**id1**, **id2**, and so on) can be used inside the handler, just like a method argument. Sometimes you never use the identifier because the type of the **Exception** gives you enough information to diagnose and respond to the exceptional condition. In that situation, you can leave the identifier out altogether as is done with the **Type3** catch block above.

The handlers must appear directly after the try block. If an exception is thrown, the exception handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled. The search for handlers stops once the catch clause is finished. Only the matching catch clause executes; it's not like a **switch** statement in which you need a **break** after each **case**.

Note that, within the try block, a number of different method calls might generate the same exception, but you need only one handler.

Supertype matching

Naturally, the catch block will match a type descended from the specified type (since inheritance is an *is-a* type relationship). So the line

```
}catch(Exception ex){ … }
```

will match any type of exception. A not uncommon mistake in Java code is an overly-general catch block above a more specific catch block, but the C# compiler detects such mistakes and will not allow this mistake.

Exceptions have a helplink

The **Exception** class contains a **string** property called **HelpLink**. This property is intended to hold a URI and the .NET Framework SDK documentation suggests that you might refer to a helpfile explaining the error. On the other hand, as we'll

discuss in Chapter 18, a URI is all you need to call a Web Service. One can imagine using **Exception.HelpLink** and a little ingenuity to develop an error-reporting system along the lines of Windows XP's that packages the context of an exception, asks the user for permission, and sends it off to a centralized server. At the server, you could parse the **Exception.StackTrace** to determine if the exception was known or a mystery and then take appropriate steps such as sending emails or pages.

Creating your own exceptions

You're not stuck using the existing C# exceptions. This is important because you'll often need to create your own exceptions to denote a special error that your library is capable of creating, but which was not foreseen when the C# exception hierarchy was created. C#'s predefined exceptions derive from SystemException, while your exceptions are expected to derive from ApplicationException.

To create your own exception class, you're forced to inherit from an existing type of exception, preferably one that is close in meaning to your new exception (this is often not possible, however). The most trivial way to create a new type of exception is just to let the compiler create the default constructor for you, so it requires almost no code at all:

```
//:c11:SimpleExceptionDemo.cs
// Inheriting your own exceptions.
using System;

class SimpleException : ApplicationException {
}

public class SimpleExceptionDemo {
  public void F() {
    Console.WriteLine(
      "Throwing SimpleException from F()");
    throw new SimpleException ();
  }
  public static void Main() {
    SimpleExceptionDemo sed =
      new SimpleExceptionDemo();
    try {
      sed.F();
    } catch (SimpleException ) {
      Console.Error.WriteLine("Caught it!");
```

```
    }
  }
} ///:~
```

When the compiler creates the default constructor, it automatically (and invisibly) calls the base-class default constructor. As you'll see, the most important thing about an exception is the class name, so most of the time an exception like the one shown above is satisfactory.

Here, the result is printed to the console *standard error* stream by writing to **System.Console.Error**. This stream can be redirected to any other **TextWriter** by calling **System.Console.SetError( )** (note that this is "asymmetric" – the **Error** property doesn't support assignment, but there's a **SetError( )**. Why would this be?).

Creating an exception class that overrides the standard constructors is also quite simple:

```
//:c11:FullConstructors.cs
using System;

class MyException : Exception {
  public MyException() : base() {}
  public MyException(string msg) : base(msg) {}
  public MyException(string msg, Exception inner) :
    base(msg, inner){}
}

public class FullConstructors {
  public static void F() {
    Console.WriteLine(
      "Throwing MyException from F()");
    throw new MyException();
  }
  public static void G()  {
    Console.WriteLine(
      "Throwing MyException from G()");
    throw new MyException("Originated in G()");
  }

  public static void H(){
    try {
      I();
```

```
    } catch (DivideByZeroException e) {
      Console.WriteLine(
        "Increasing abstraction level");
      throw new MyException("Originated in H()", e);
    }
  }

  public static void I(){
    Console.WriteLine("This'll cause trouble");
    int y = 0;
    int x = 1 / y;
  }

  public static void Main() {
    try {
      F();
    } catch (MyException e) {
      Console.Error.WriteLine(e.StackTrace);
    }
    try {
      G();
    } catch (MyException e) {
      Console.Error.WriteLine(e.Message);
    }
    try {
      H();
    } catch (MyException e) {
      Console.Error.WriteLine(e.Message);
      Console.Error.WriteLine("Inner exception: "
        + e.InnerException);
      Console.Error.WriteLine("Source: " + e.Source);
      Console.Error.WriteLine("TargetSite: "
        + e.TargetSite);
    }
  }
} ///:~
```

The code added to **MyException** is small—the addition of three constructors that define the way **MyException** is created. The base-class constructor is explicitly invoked by using the **: base** keyword.

The output of the program is:

```
Throwing MyException from F()
   at FullConstructors.F()
   at FullConstructors.Main()
Throwing MyException from G()
Originated in G()
This'll cause trouble
Increasing abstraction level
Originated in H()
Inner exception: System.DivideByZeroException: Attempted to
divide by zero.
   at FullConstructors.I()
   at FullConstructors.H()
Source: FullConstructors
TargetSite: Void H()
```

You can see the absence of the detail message in the **MyException** thrown from **F( )**. The block that catches the exception thrown from **F( )** shows the stack trace all the way to the origin of the exception. This is probably the most helpful property in Exception and is a great aid to debugging.

When **H( )** executes, it calls **I( )**, which attempts an illegal arithmetic operation. The attempt to divide by zero throws a **DivideByZeroException** (demonstrating the truth of the previous statement about the type name being the most important thing). **H( )** catches the **DivideByZeroException**, but increases the abstraction level by wrapping it in a **MyException**. Then, when the **MyException** is caught in **Main( )**, we can see the inner exception and its origin in **I( )**.

The **Source** property contains the name of the assembly that threw the exception, while the **TargetSite** property returns a handle to the method that threw the exception. **TargetSite** is appropriate for sophisticated reflection-based exception diagnostics and handling.

The process of creating your own exceptions can be taken further. You can add extra constructors and members:

```
//:c11:ExtraFeatures.cs
// Further embellishment of exception classes.
using System;

class MyException2 : Exception {
  int errorCode;
  public int ErrorCode{
```

```
    get { return errorCode;}
  }

  public MyException2() : base(){}

  public MyException2(string msg) : base(msg) {}

  public MyException2(string msg, int errorCode) :
    base(msg) {
    this.errorCode = errorCode;
  }
}

public class ExtraFeatures {
  public static void F() {
    Console.WriteLine(
      "Throwing MyException2 from F()");
    throw new MyException2();
  }
  public static void G()  {
    Console.WriteLine(
      "Throwing MyException2 from G()");
    throw new MyException2("Originated in G()");
  }
  public static void H() {
    Console.WriteLine(
      "Throwing MyException2 from H()");
    throw new MyException2(
      "Originated in H()", 47);
  }
  public static void Main(String[] args) {
    try {
      F();
    } catch (MyException2 e) {
      Console.Error.WriteLine(e.StackTrace);
    }
    try {
      G();
    } catch (MyException2 e) {
      Console.Error.WriteLine(e.StackTrace);
    }
```

```
    try {
      H();
    } catch (MyException2 e) {
      Console.Error.WriteLine(e.StackTrace);
      Console.Error.WriteLine("e.ErrorCode = "
        + e.ErrorCode);
    }
  }
} ///:~
```

A property **ErrorCode** has been added, along with an additional constructor that sets it. The output is:

```
Throwing MyException2 from F()
   at ExtraFeatures.F() in
D:\tic\exceptions\ExtraFeatures.cs:line 23
   at ExtraFeatures.Main(String[] args) in C:\Documents and
Settings\larry\My Documents\ExtraFeatures.cs:line 38
Throwing MyException2 from G()
   at ExtraFeatures.G() in
D:\tic\exceptions\ExtraFeatures.cs:line 28
   at ExtraFeatures.Main(String[] args) in
D:\tic\exceptions\ExtraFeatures.cs:line 43
Throwing MyException2 from H()
   at ExtraFeatures.H() in
D:\tic\exceptions\ExtraFeatures.cs:line 33
   at ExtraFeatures.Main(String[] args) in C:\Documents and
Settings\larry\My Documents\ExtraFeatures.cs:line 48
e.ErrorCode = 47
```

Since an exception is just another kind of object, you can continue this process of embellishing the power of your exception classes. An error code, as illustrated here, is probably not very useful, since the type of the **Exception** gives you a good idea of the "what" of the problem. More interesting would be to embed a clue as to the "how" of retrying or cleanup – some kind of object that encapsulated the context of the broken contract. Keep in mind, however, that the best design is the one that throws exceptions rarely and that the best programmer is the one whose work delivers the most benefit to the customer, not the one who comes up with the cleverest solution to what to do when things go wrong!

C#'s lack of checked exceptions

Some languages, notably Java, require a method to list recoverable exceptions it may throw. Thus, in Java, reading data from a stream is done with a method that is declared as `int read() throws IOException` while the equivalent method in C# is simply `int read()`. This does not mean that C# somehow avoids the various unforeseeable circumstances that can ruin a read, nor even that they are necessarily less likely to occur in C#. If you look at the documentation for `System.IO.Stream.Read()` you'll see that it can throw, yes, an **IOException**.

The effect of including a list of exceptions in a method's declaration is that at compile-time, if the method is used, the compiler can assure that the **Exception** is either handled or passed on. Thus, in languages like Java, the exception is explicitly part of the method's signature – "Pass me parameters of type such and so and I'll return a value of a certain type. However, if things go awry, I may also throw these particular types of **Exception**." Just as the compiler can enforce that a method that takes an **int** and returns a **string** is not passed a **double** or used to assign to a **float** so too does the compiler enforce the exception specification.

One big problem with checked exceptions is that it locks an implementer into the exception specification of the base class method. In Java, if you declare a class and method:

```
interface Movie{
   void Enjoy() throws PeopleTalkingException{…}
}
```

implementations of **Movie.Enjoy( )** may throw no exceptions, but the only checked exceptions they *may* throw are of type **PeopleTalkingException**. The good side of this is that any code that works with the **Movie** interface and that catches **PeopleTalkingException**s is guaranteed to continue to work, no matter how the **Movie** interface is implemented. But the down side is that sometimes the assumption about what constitutes a valid exception is wrong: Let's say that you want to implement a **HomeMovie** where people can talk all they want, but when the phone rings it is an exceptional circumstance. In Java, you are forced either to rewrite the base interface's exception specification, inherit **PhoneRingingException** from **PeopleTalkingException**, or forego the supposed benefits of checked exceptions.

Checked exceptions such as in Java are not intended to deal with precondition violations, which are by far the most common cause of exceptional conditions. A

precondition violation (calling a method with an improper parameter, calling a state-specific method on an object that's not in the required state) is, by definition, the result of a programming error. Retries are, at best, useless in such a situation (at worst, the retry will work and thereby allow the programming error to go unfixed!). So Java has another type of exception that is unchecked.

In practice what happens is that while programmers are generally accepting of strong type-checking when it comes to parameters and return values, the value of strongly typed exceptions is not nearly as evident in real-world practice. There are too many low-level things that can go wrong (failures of files and networks and RAM and so forth) and many programmers do not see the benefit of creating an abstraction hierarchy as they deal with all failures in a generic manner. And the different intent of checked and unchecked exceptions is confusing to many developers.

And thus one sees a great deal of Java code in two equally bad forms: meaningless propagation of low-abstraction exceptions (Web services that are declared as throwing IOExceptions) and "make it compile" hacks where methods are declared as "throws Exception" (in other words, saying "I can throw anything I darn well please."). Worse, though, it's not uncommon to see the very worst possible "solution," which is to catch and ignore the exception, all for the sake of getting a clean compile.

Theoretically, if you're going to have a strongly typed language, you can make an argument for exceptions being part of the method signature. Pragmatically, though, the prevalence of bad exception-handling code in Java argues for C#'s approach, which is essentially that the burden is on the programmer to know to place error-handling code in the appropriate places.

Catching any exception

It is possible to create a handler that catches any type of exception. You do this by catching the base-class exception type **Exception**:

```
catch(Exception e) {
  Console.Error.WriteLine("Caught an exception");
}
```

This will catch any exception, so if you use it you'll want to put it at the *end* of your list of handlers to avoid preempting any exception handlers that might otherwise follow it.

Since the **Exception** class is the base of all the exception classes, you don't get much specific information about the specific problem. You do, however, get some

methods from **object** (everybody's base type). The one that might come in handy for exceptions is **GetType( )**, which returns an object representing the class of **this**. You can in turn read the **Name** property of this **Type** object. You can also do more sophisticated things with **Type** objects that aren't necessary in exception handling. **Type** objects will be studied later in this book.

Rethrowing an exception

Sometimes you'll want to rethrow the exception that you just caught, particularly when you use **catch(Exception)** to catch any exception. Since you already have the reference to the current exception, you can simply rethrow that reference by using **throw** again:

```
catch(Exception e) {
  Console.Error.WriteLine("An exception was thrown");
  throw;
}
```

Note that unlike the first time an exception is thrown, a rethrow does not require an explicit reference to an exception; rather, the rethrow will use the exception that was passed in to the exception block. Rethrowing an exception causes the exception to go to the exception handlers in the next-higher context. Any further **catch** clauses for the same **try** block are still ignored. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type can extract all the information from that object.

Elevating the abstraction level

Usually when catching exceptions and then propagating them outward, you should elevate the abstraction level of the caught **Exception**. For instance, at the business-logic level, all you may care about is that "the charge didn't go through." You'll certainly want to preserve the information of the less-abstract **Exception** for debugging purposes, but for logical purposes, you want to deal with all problems equally.

```
//:c11:Rethrow.cs
using System;

class TransactionFailureException :
  ApplicationException {
  public TransactionFailureException(Exception e) :
  base("Logical failure caused by low-level "
    + "exception", e){
```

```
  }
}

class Transaction {
  Random r = new Random();
  public void Process(){
    try {
      if (r.NextDouble() > 0.3) {
        throw new ArithmeticException();
      } else {
        if (r.NextDouble() > 0.5) {
          throw new FormatException();
        }
      }
    } catch (Exception e) {
      TransactionFailureException tfe =
        new TransactionFailureException(e);
      throw tfe;
    }
  }
}


class BusinessLogic {
  Transaction myTransaction = new Transaction();

  public void DoCharge(){
    try {
      myTransaction.Process();
      Console.WriteLine("Transaction ok");
    } catch (TransactionFailureException tfe) {
      Console.WriteLine(tfe.Message);
      Console.Error.WriteLine(tfe.InnerException);
    }
  }

  public static void Main(){
    BusinessLogic bl = new BusinessLogic();
    for (int i = 0; i < 10; i++) {
      bl.DoCharge();
    }
```

```
    }
} ///:~
```

In this example, the class **Transaction** has an exception class that is at its same level of abstraction in **TransactionFailureException**. The **try...catch(Exception e**) construct in **Transaction.Process( )** makes for a nice and explicit contract: "I try to return void, but if anything goes awry in my processing, I may throw a **TransactionFailedException**." In order to generate some exceptions, we use a random number generator to throw different types of low-level exceptions in **Transaction.Process( )**.

All exceptions are caught in **Transaction.Process( )**'s catch block, where they are placed "inside" a new **TransactionFailureException** using that type's overridden constructor that takes an exception and creates a generic "Logical failure caused by low-level exception" message. The code then throws the newly created **TransactionFailureException**, which is in turn caught by **BusinessLogic.DoCharge( )**'s **catch(TransactionFailureException tfe)** block. The higher-abstraction exception's message is printed to the console, while the lower-abstraction exception is sent to the Error stream (which is also the console, but the point is that there is a separation between the two levels of abstraction. In practice, the higher-abstraction exception would be used for business logic choices and the lower-abstraction exception for debugging).

Standard C# exceptions

The C# class **Exception** describes anything that can be thrown as an exception. There are two general types of **Exception** objects ("types of" = "inherited from"). **SystemException** represents exceptions in the System namespace and its descendants (in other words, .NET's standard exceptions). Your exceptions by convention should extend from **ApplicationException**.

If you browse the .NET documentation, you'll see that each namespace has a small handful of exceptions that are at a level of abstraction appropriate to the namespace. For instance, System.IO has an **InternalBufferOverflowException**, which is pretty darn low-level, while System.Web.Services.Protocols has **SoapException**, which is pretty darn high-level. It's worth browsing through these exceptions once to get a feel for the various exceptions, but you'll soon see that there isn't anything special between one exception and the next except for the name. The basic idea is that the name of the exception represents the problem that occurred, and the exception name is intended to be relatively self-explanatory.

Performing cleanup with finally

There's often some piece of code that you want to execute whether or not an exception is thrown within a **try** block. This usually pertains to some operation other than memory recovery (since that's taken care of by the garbage collector). To achieve this effect, you use a **finally** clause at the end of all the exception handlers. The full picture of an exception handling section is thus:

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

In **finally** blocks, you can use control flow statements **break, continue,** or **goto** only for loops that are entirely inside the **finally** block; you cannot perform a jump out of the **finally** block. Similarly, you can not use **return** in a **finally** block. Violating these rules will give a compiler error.

To demonstrate that the **finally** clause always runs, try this program:

```
//:c11:AlwaysFinally.cs
// The finally clause is always executed.
using System;

class ThreeException : ApplicationException { }

public class FinallyWorks {
  static int count = 0;
  public static void Main() {
    while (true) {
      try {
        if (count++ < 3) {
          throw new ThreeException();
```

```
        }
        Console.WriteLine("No exception");
      } catch (ThreeException ) {
        Console.WriteLine("ThreeException");
      } finally {
        Console.Error.WriteLine("In finally clause");
        //! if(count == 3) break; <- Compiler error
      }
      if (count > 3)
        break;
    }
  }
} ///:~
```

This program also gives a hint for how you can deal with the fact that exceptions in C# do not allow you to resume back to where the exception was thrown, as discussed earlier. If you place your **try** block in a loop, you can establish a condition that must be met before you continue the program. You can also add a **static** counter or some other device to allow the loop to try several different approaches before giving up. This way you can build a greater level of robustness into your programs.

The output is:

```
ThreeException
In finally clause
ThreeException
In finally clause
ThreeException
In finally clause
No exception
In finally clause
```

Whether an exception is thrown or not, the **finally** clause is always executed.

What's finally for?

Since C# has a garbage collector, releasing memory is virtually never a problem. So why do you need **finally**?

finally is necessary when you need to set something *other* than memory back to its original state. This is some kind of cleanup like an open file or network connection, something you've drawn on the screen, or even a switch in the outside world, as modeled in the following example:

```
//:c11:WhyFinally.cs
// Why use finally?
using System;

class Switch {
  bool state = false;
  public bool Read{
    get { return state;}
    set { state = value;}
  }

  public void On(){ state = true;}
  public void Off(){ state = false;}
}
class OnOffException1 : Exception {
}
class OnOffException2 : Exception {
}

public class OnOffSwitch {
  static Switch sw = new Switch();
  static void F() {}
  public static void Main() {
    try {
      sw.On();
      // Code that can throw exceptions...
      F();
      sw.Off();
    } catch (OnOffException1 ) {
      Console.WriteLine("OnOffException1");
      sw.Off();
    } catch (OnOffException2 ) {
      Console.WriteLine("OnOffException2");
      sw.Off();
    }
  }
} ///:~
```

The goal here is to make sure that the switch is off when **Main( )** is completed, so **sw.Off( )** is placed at the end of the try block and at the end of each exception handler. But it's possible that an exception could be thrown that isn't caught here,

so **sw.Off( )** would be missed. However, with **finally** you can place the cleanup code from a try block in just one place:

```
//:c11:WhyFinally2.cs
// Why use finally?
using System;

class Switch {
  bool state = false;
  public bool Read{
    get { return state;}
    set { state = value;}
  }

  public void On(){ state = true;}
  public void Off(){ state = false;}
}
class OnOffException1 : Exception {
}
class OnOffException2 : Exception {
}

public class OnOffSwitch {
  static Switch sw = new Switch();
  static void F() {}
  public static void Main() {
    try {
      sw.On();
      // Code that can throw exceptions...
      F();
    } catch (OnOffException1 ) {
      Console.WriteLine("OnOffException1");
    } catch (OnOffException2 ) {
      Console.WriteLine("OnOffException2");
    } finally {
      sw.Off();
    }
  }
} ///:~
```

Here the **sw.Off( )** has been moved to just one place, where it's guaranteed to run no matter what happens.

Even in cases in which the exception is not caught in the current set of **catch** clauses, **finally** will be executed before the exception handling mechanism continues its search for a handler at the next higher level:

```
//:c11:NestedFinally.cs
// Finally is always executed.
using System;

class FourException : ApplicationException {}

public class AlwaysFinally {
  public static void Main() {
    Console.WriteLine(
      "Entering first try block");
    try {
      Console.WriteLine(
        "Entering second try block");
      try {
        throw new FourException();
      } finally {
        Console.WriteLine(
          "finally in 2nd try block");
      }
    } catch (FourException ) {
      Console.WriteLine(
        "Caught FourException in 1st try block");
    } finally {
      Console.WriteLine(
        "finally in 1st try block");
    }
  }
} ///:~
```

The output for this program shows you what happens:

```
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
```

Finally **and** using

Way back in Chapter #initialization and cleanup#, we discussed C#'s **using** blocks. Now we can finally explain how it works. Consider this code, which uses inheritance, upcasting, and a **try...finally** block to ensure that cleanup happens:

```
//:c11:UsingCleanup2.cs
using System;

class UsingCleanup : IDisposable {
  public static void Main(){
    try {
      IDisposable uc = new UsingCleanup();
      try {
        throw new NotImplementedException();
      } finally {
        uc.Dispose();
      }
    } catch (Exception ) {
      Console.WriteLine("After disposal");
    }
  }

  UsingCleanup(){
    Console.WriteLine("Constructor called");
  }

  public void Dispose(){
    Console.WriteLine("Dispose called");
  }

  ~UsingCleanup(){
    Console.WriteLine("Destructor called");
  }
}///:~
```

You should not be surprised at the output:

```
Constructor called
Dispose called
After disposal
Destructor called
```

Changing the **Main( )** method to:

```
    public static void Main(){
        try {
            UsingCleanup uc = new UsingCleanup();
            using(uc){
                throw new NotImplementedException();
            }
        }catch(Exception ){
            Console.WriteLine("After disposal");
        }
    }
```

produces the exact same output. In fact, the **using** keyword is just "syntactic sugar" that wraps an **IDisposable** subtype in a **try...finally** block! Behind the scenes, the exact same code is generated, but the **using** block is terser.

Pitfall: the lost exception

In general, C#'s exception implementation is quite outstanding, but unfortunately there's a flaw. Although exceptions are an indication of a crisis in your program and should never be ignored, it's possible for an exception to simply be lost. This happens with a particular configuration using a **finally** clause:

```
//:c11:LostException.cs
// How an exception can be lost.
using System;

class VeryImportantException : Exception {
}

class HoHumException : Exception {
}

public class LostMessage {
  void F() {
    throw new VeryImportantException();
  }
  void Dispose()  {
    throw new HoHumException();
  }
  public static void Main(){
```

```
    try {
      LostMessage lm = new LostMessage();
      try {
        lm.F();
      } finally {
        lm.Dispose();
      }
    } catch (Exception e) {
      Console.WriteLine(e);
    }
  }
} ///:~
```

The output is:

```
HoHumException: Exception of type HoHumException was
thrown.
   at LostMessage.Dispose()
   at LostMessage.Main()
```

You can see that there's no evidence of the **VeryImportantException**, which is simply replaced by the **HoHumException** in the **finally** clause. This is a rather serious pitfall, since it means that an exception can be completely lost, and in a far more subtle and difficult-to-detect fashion than the example above. In contrast, C++ treats the situation in which a second exception is thrown before the first one is handled as a dire programming error. To avoid this possibility, it is a good idea to wrap all your work inside a **finally** block in a **try...catch(Exception):**

```
//:c11:CarefulFinally.cs
using System;

class VeryImportantException : Exception {
}

class HoHumException : Exception {
}

public class LostMessage {
  void F() {
    throw new VeryImportantException();
  }
  void Dispose()  {
```

```
    throw new HoHumException();
  }
  public static void Main(){
    try {
      LostMessage lm = new LostMessage();
      try {
        lm.F();
      } finally {
        try {
          lm.Dispose();
        } catch (Exception e) {
          Console.WriteLine(e);
        }
      }
    } catch (Exception e) {
      Console.WriteLine(e);
    }
  }
} ///:~
```

Produces the desired output:

```
HoHumException: Exception of type HoHumException was
thrown.
   at LostMessage.Dispose()
   at LostMessage.Main()
VeryImportantException: Exception of type
VeryImportantException was thrown.
   at LostMessage.F()
   at LostMessage.Main()
```

Constructors

When writing code with exceptions, it's particularly important that you always ask, "If an exception occurs, will this be properly cleaned up?" Most of the time you're fairly safe, but in constructors there's a problem. The constructor puts the object into a safe starting state, but it might perform some operation—such as opening a file—that doesn't get cleaned up until the user is finished with the object and calls a special cleanup method. If you throw an exception from inside a constructor, these cleanup behaviors might not occur properly. This means that you must be especially diligent while you write your constructor.

Since you've just learned about **finally**, you might think that it is the correct solution. But it's not quite that simple, because **finally** performs the cleanup code *every time,* even in the situations in which you don't want the cleanup code executed until the cleanup method runs. Thus, if you do perform cleanup in **finally**, you must set some kind of flag when the constructor finishes normally so that you don't do anything in the **finally** block if the flag is set. Because this isn't particularly elegant (you are coupling your code from one place to another), it's best if you try to avoid performing this kind of cleanup in **finally** unless you are forced to.

In the following example, a class called **InputFile** is created that opens a file and allows you to read it one line (converted into a **string**) at a time. It uses the classes **FileReader** and **BufferedReader** from the Java standard I/O library that will be discussed in Chapter 12, but which are simple enough that you probably won't have any trouble understanding their basic use:

```
//:c11:ConstructorFinally.cs
// Paying attention to exceptions
// in constructors.
using System;
using System.IO;

namespace Cleanup{
  internal class InputFile : IDisposable {
    private StreamReader inStream;
    internal InputFile(string fName) {
      try {
        inStream =
        new StreamReader(
          new FileStream(
            fName, FileMode.Open));
        // Other code that might throw exceptions
      } catch (FileNotFoundException e) {
        Console.Error.WriteLine(
          "Could not open " + fName);
        // Wasn't open, so don't close it
        throw e;
      } catch (Exception e) {
        // All other exceptions must close it
        try {
          inStream.Close();
        } catch (IOException ) {
```

```
          Console.Error.WriteLine(
            "in.Close() unsuccessful");
        }
        throw e; // Rethrow
      } finally {
        // Don't close it here!!!
      }
    }
    internal string ReadLine() {
      string s;
      try {
        s = inStream.ReadLine();
      } catch (IOException ) {
        Console.Error.WriteLine(
          "ReadLine() unsuccessful");
        s = "failed";
      }
      return s;
    }
    public void Dispose() {
      try {
        inStream.Close();
      } catch (IOException ) {
        Console.Error.WriteLine(
          "in.Close() unsuccessful");
      }
    }
  }

  public class Cleanup {
    public static void Main() {
      try {
        InputFile inFile =
          new InputFile("Cleanup.cs");
        using(inFile){
          String s;
          int i = 1;
          while ((s = inFile.ReadLine()) != null)
            Console.WriteLine(
              ""+ i++ + ": " + s);
        }
```

```
      } catch (Exception e) {
        Console.Error.WriteLine("Caught in Main");
        Console.Error.WriteLine(e.StackTrace);
      }
    }
  }
} ///:~
```

The constructor for **InputFile** takes a **string** argument, which is the name of the file you want to open. Inside a **try** block, it creates a **FileStream** using the filename. A **FileStream** isn't particularly useful for text until you turn around and use it to create a **StreamReader** that can deal with more than one character at a time.

If the **FileStream** constructor is unsuccessful, it throws a **FileNotFoundException**, which must be caught separately because that's the one case in which you don't want to close the file since it wasn't successfully opened. Any *other* catch clauses must close the file because it *was* opened by the time those catch clauses are entered. (Of course, this is trickier if more than one method can throw a **FileNotFoundException**. In that case, you might want to break things into several **try** blocks.) The **Close( )** method might throw an exception so it is tried and caught even though it's within the block of another **catch** clause—it's just another pair of curly braces to the C# compiler. After performing local operations, the exception is rethrown, which is appropriate because this constructor failed, and you wouldn't want the calling method to assume that the object had been properly created and was valid.

In this example, which doesn't use the aforementioned flagging technique, the **finally** clause is definitely *not* the place to **Close( )** the file, since that would close it every time the constructor completed. Since we want the file to be open for the useful lifetime of the **InputFile** object this would not be appropriate.

The **ReadLine( )** method returns a **string** containing the next line in the file. It calls **StreamReader.ReadLine( ),** which can throw an exception, but that exception is caught so **ReadLine( )** doesn't throw any exceptions.

The **Dispose( )** method must be called when the **InputFile** is finished with. This will release the system resources (such as file handles) that are used by the **StreamReader** and/or **FileStream** objects. You don't want to do this until you're finished with the **InputFile** object, at the point you're going to let it go. You might think of putting such functionality into a destructor method, but as mentioned in Chapter 5 you can't always be sure when the destructor will be

called (even if you *can* be sure that it will be called, all you know about *when* is that it's sure to be called before the process ends).

In the **Cleanup** class, an **InputFile** is created to open the same source file that creates the program, the file is read in a line at a time, and line numbers are added. The **using** keyword is used to ensure that **InputFile.Dispose( )** is called. All exceptions are caught generically in **Main( )**, although you could choose greater granularity.

One of the benefits of this example is to show you why exceptions are introduced at this point in the book—you can't do basic I/O without using exceptions. Exceptions are so integral to programming in C# that you can accomplish only so much without knowing how to work with them.

Exception matching

When an exception is thrown, the exception handling system looks through the "nearest" handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn't require a perfect match between the exception and its handler. A derived-class object will match a handler for the base class, as shown in this example:

```
//:c11:Sneeze.cs
// Catching exception hierarchies.
using System;

class Annoyance : Exception {
}
class Sneeze : Annoyance {
}

public class Human {
  public static void Main() {
    try {
      throw new Sneeze();
    } catch (Sneeze ) {
      Console.Error.WriteLine("Caught Sneeze");
    } catch (Annoyance ) {
      Console.Error.WriteLine("Caught Annoyance");
    }
  }
```

```
} ///:~
```

The **Sneeze** exception will be caught by the first **catch** clause that it matches—which is the first one, of course. However, if you remove the first catch clause, leaving only:

```
try {
  throw new Sneeze();
} catch(Annoyance) {
  Console.Error.WriteLine("Caught Annoyance");
}
```

The code will still work because it's catching the base class of **Sneeze**. Put another way, **catch(Annoyance e)** will catch an **Annoyance** *or any class derived from it*. This is useful because if you decide to add more derived exceptions to a method, then the client programmer's code will not need changing as long as the client catches the base class exceptions.

If you try to "mask" the derived-class exceptions by putting the base-class catch clause first, like this:

```
try {
  throw new Sneeze();
} catch(Annoyance a) {
  Console.Error.WriteLine("Caught Annoyance");
} catch(Sneeze s) {
  Console.Error.WriteLine("Caught Sneeze");
}
```

the compiler will give you an error message, since it sees that the **Sneeze** catch-clause can never be reached.

Exception guidelines

Use exceptions to:

1. Fix the problem and call the method that caused the exception again.
2. Patch things up and continue without retrying the method.
3. Calculate some alternative result instead of what the method was supposed to produce.
4. Do whatever you can in the current context and rethrow the *same* exception to a higher context.

5. Do whatever you can in the current context and throw a *different* exception to a higher context.
6. Terminate the program.
7. Simplify. (If your exception scheme makes things more complicated, then it is painful and annoying to use.)
8. Make your library and program safer. (This is a short-term investment for debugging, and a long-term investment for application robustness.)

Summary

Improved error recovery is one of the most powerful ways that you can increase the robustness of your code. Error recovery is a fundamental concern for every program you write, but it's especially important in C#, where one of the primary goals is to create program components for others to use. *To create a robust system, each component must be robust.*

Exceptions are not terribly difficult to learn, and are one of those features that provide immediate and significant benefits to your project.

Exercises

1. Create a class with a **Main( )** that throws an object of class **Exception** inside a **try** block. Give the constructor for **Exception** a **string** argument. Catch the exception inside a **catch** clause and print the **string** argument. Add a **finally** clause and print a message to prove you were there.
2. Create your own exception class. Write a constructor for this class that takes a **string** argument and stores it inside the object with a **string** reference. Write a method that prints out the stored **string**. Create a **try-catch** clause to exercise your new exception.
3. Write a class with a method that throws an exception of the type created in the previous exercise. Try compiling it without an exception specification to see what the compiler says. Add the appropriate exception specification. Try out your class and its exception inside a try-catch clause.
4. Define an object reference and initialize it to **null**. Try to call a method through this reference. Now wrap the code in a **try-catch** clause to catch the exception.

5. Create a class with two methods, **F( )** and **G( )**. In **G( )**, throw an exception of a new type that you define. In **F( )**, call **G( )**, catch its exception and, in the **catch** clause, throw a different exception (of a second type that you define). Test your code in **Main( )**.

6. Create three new types of exceptions. Write a class with a method that throws all three. In **Main( )**, call the method but only use a single **catch** clause that will catch all three types of exceptions.

7. Write code to generate and catch an **IndexOutOfRangeException**.

8. Create your own resumption-like behavior using a **while** loop that repeats until an exception is no longer thrown.

9. Create a three-level hierarchy of exceptions. Now create a base-class **A** with a method that throws an exception at the base of your hierarchy. Inherit **B** from **A** and override the method so it throws an exception at level two of your hierarchy. Repeat by inheriting class **C** from **B**. In **Main( )**, create a **C** and upcast it to **A**, then call the method.

10. Demonstrate that a derived-class constructor cannot catch exceptions thrown by its base-class constructor.

11. Add a second level of exception loss to **LostMessage.cs** so that the **HoHumException** is itself replaced by a third exception.

12: I/O in C#

Creating a good input/output (I/O) system is one of the more difficult tasks for the language designer.

This is evidenced by the number of different approaches. The challenge seems to be in covering all eventualities. Not only are there different sources and sinks of I/O that you want to communicate with (files, the console, network connections), but you need to talk to them in a wide variety of ways (sequential, random-access, buffered, binary, character, by lines, by words, etc.).

The .NET library designers attacked this problem by creating lots of classes. In fact, there are so many classes for .NET's I/O system that it can be intimidating at first (ironically, the design actually prevents an explosion of classes). As a result there are a fair number of classes to learn before you understand enough of .NET's I/O picture to use it properly.

File, Directory, **and** Path

Before getting into the classes that actually read and write data to streams, we'll look at the utility classes that assist you in handling file directory issues. These utility classes consist of three classes that have just static methods: **Path**, **Directory**, and **File**. These classes have somewhat confusing names in that there's no correspondence between object instances and items within the file-system (indeed, you can't instantiate these classes – their constructors are not public).

A directory lister

Let's say you want to list the names of the files in the directory. This is easily done with the static **Directory.GetFiles( )** method, as is shown in this sample:

```
//c12:FList.cs
//Displays directory listing
using System.IO;

public class FList{
   public static void Main(string[] args){
      string dirToRead = ".";
      string pattern = "*";
```

```
      if(args.Length > 0){
         dirToRead = args[0];
      }
      if(args.Length > 1){
         pattern = args[1];
      }
      string[] fPaths =
         Directory.GetFiles(dirToRead, pattern);
      foreach(string fPath in fPaths){
         FileInfo fInfo = new FileInfo(fPath);
         Console.WriteLine(
            "Path = {0} Filename: {1} ext: {2} "
            + "touch: {3} size: {4}",
            fPath, fInfo.Name, fInfo.Extension,
            fInfo.LastWriteTime, fInfo.Length);
         Console.WriteLine(fName);
      }
   }
}///:~
```

When run with no arguments, this returns all the names in the current directory; another directory can be specified with the first argument, and a standard DOS file-name pattern with the second. An overload of the **Directory.GetFiles( )** method takes just a single string, which is equivalent to calling **Directory.GetFiles(dirString, "*").**

The method **Directory.GetFiles( )** is a little poorly named, it's really returned strings that represent the paths to files (so perhaps it would have been better named **GetFilePaths( )**). To get information on the corresponding file, the path is passed in to a **FileInfo** constructor. The **FileInfo** and related **DirectoryInfo** classes encapsulate what the file system knows – things like size, time of creation and last edit, attributes such as being ReadOnly, etc.

Checking for and creating directories

The **FList** program above returned only paths to files, not to directories. **Directory.GetFileSystemEntries( )** returns paths to both files and directories, while **Directory.GetDirectories( )** returns paths to the subdirectories of the given directory.

```
//:c12:DirList.cs
//Displays listing of subdirectories
using System;
```

```
using System.IO;

public class DirList {
  public static void Main(string[] args){
    string dirToRead = ".";
    string pattern = "*";
    if (args.Length > 0) {
      dirToRead = args[0];
    }
    if (args.Length > 1) {
      pattern = args[1];
    }
    string[] subdirs =
      Directory.GetDirectories(dirToRead, pattern);
    foreach(string subdir in subdirs){
      DirectoryInfo dInfo =
      new DirectoryInfo(subdir);
      Console.WriteLine(
        "Path = {0} Created: {1} Accessed: {2} "
        + " Written to {3} ",
        subdir, dInfo.CreationTime,
        dInfo.LastAccessTime, dInfo.LastWriteTime);
    }
  }
}///:~
```

In addition to getting information on files and directories, the **File** and **Directory** classes contain methods for creating, deleting, and moving filesystem entities. This example shows how to create new subdirectories, files, and delete directories:

```
//:c12:FileManip.cs
//Demonstrates basic filesystem manipulation
using System;
using System.IO;

class FileManip {
  public static void Main(){
    string curPath =
      Directory.GetCurrentDirectory();
    DirectoryInfo curDir =
      new DirectoryInfo(curPath);
```

```
    string curName = curDir.Name;
    char[] chars = curName.ToCharArray();
    Array.Reverse(chars);
    string revName = new String(chars);
    if (Directory.Exists(revName)) {
      Console.WriteLine("Deleting dir " + revName);
      Directory.Delete(revName, true);
    } else {
      Console.WriteLine("Making dir " + revName);
      Directory.CreateDirectory(revName);
      string fName = "./" + revName + "/Foo.file";
      File.Create(fName);
    }
  }
}///:~
```

First, we use **Directory.GetCurrentDirectory( )** to retrieve the current path; the same data is also available as **Environment.CurrentDirectory**. To get the current directory's name, we use the **DirectoryInfo** class and its **Name** property. The name of our new directory is the current directory's name in reverse. The first time this program runs, **Directory.Exists( )** will return false (unless you happen to run this program in a directory with a reversed-name subdirectory already there!). In that case, we use **Directory.CreateDirectory( )** and **File.Create( )** (note the slight inconsistency in naming) to create a new subdirectory and a file. If you check, you'll see that "Foo.file" is of length 0 – **File.Create( )** works at the filesystem level, not at the level of actually initializing the file with useful data.

The second time **FileManip** is run, the **Exists( )** method will return true and **Directory.Delete( )** deletes both the directory *and all its contents, including files and subdirectories*. If you don't want this highly dangerous behavior, either pass in a **false** value, or use the overloaded **Directory.Delete(string)** which doesn't take a bool and which will throw an **IOException** if called on a non-empty directory.

Isolated stores

Some of the most common file-oriented tasks are associated with a single user: Preferences should be set to individual users, security dictates that there be restrictions on arbitrary file manipulation by components downloaded off the Web, etc. In these scenarios, the .NET Framework provides for *isolated storage*. An isolated store is a virtual file system within a *data compartment*. A data compartment is based on the user, assembly, and perhaps other aspects of the

code's identity (e.g., its signature). Isolated storage is for those situations when you don't need or want database-level security and control; isolated stores end up as files on the hard drive and while operating-system restrictions may prevent them from being casually available, it's not appropriate to use isolated storage for high-value data.

Getting an isolated store for the current assembly is straightforward if wordy, one uses a static method called **GetUserStoreForAssembly( )** in the **IsolatedStorageFile** class:

```
//:c12:IsoStore.cs
using System;
using System.IO.IsolatedStorage;

class IsoStore {
  public static void Main(){
    IsolatedStorageFile isf =
    IsolatedStorageFile.GetUserStoreForAssembly();
    Console.WriteLine(
      "Assembly identity {0} \n" +
      "CurrentSize {1} \n" +
      "MaximumSize {2} \n" +
      "Scope {3} \n",
      isf.AssemblyIdentity, isf.CurrentSize,
      isf.MaximumSize, isf.Scope);
  }
}///:~
```

First, we have to specify that we're using the **System.IO.IsolatedStorage** namespace. After we create the isolated store, we print some of its attributes to the console. A typical run looks like this:

```
Assembly identity <System.Security.Policy.Url version="1">
   <Url>file://D:/tic/chap11/IsoStore.exe</Url>
</System.Security.Policy.Url>

CurrentSize 0
MaximumSize 9223372036854775807
Scope User, Assembly
```

Because we've not done any digital signing (see Chapter 13), the *identity* of the assembly this is being run from is simply the name of the assembly as a URL. The store consumes no space currently and would be allowed to consume as much as

9GB. The store that we've got a handle on is associated with the user and assembly's identity; if we changed either of those, we'd get a different isolated store.

The **IsolatedFileStore** is essentially a virtual file system, but unfortunately it does not support the general **System.IO** classes such as **Directory**, **DirectoryInfo**, **File**, and **FileInfo**. Rather, the **IsolatedFileStore** class has static methods **GetDirectoryNames( )** and **GetFileNames( )** which correspond to **Directory.GetDirectories( )** and **Directory.GetFileNames( )**. This is quite clumsy, as one cannot use objects to traverse down a tree of directories (as one can do with the **Directory** class), but rather must perform string manipulation to construct paths within the isolated store. Hopefully future versions of the framework will move towards consistency with the **System.IO** namespaces.

Input and output

I/O libraries often use the abstraction of a *stream*, which represents any data source or sink as an object capable of producing or receiving pieces of data. The stream hides the details of what happens to the data inside the actual I/O device.

The C# library classes for I/O are divided by input and output, as you can see by examining the online help reference to the .NET Framework. By inheritance, everything derived from the **Stream** class has basic methods called Read(), ReadByte(), Write(), and WriteByte() for reading and writing arrays and single bytes. However, you won't generally use these methods; they exist so that other classes can use them—these other classes provide a more useful interface. Thus, you'll rarely create an object for input or output by using a single class, but instead will layer multiple objects together to provide your desired functionality. The fact that you create more than one object to create a single resulting stream is the primary reason that .NET's IO library is confusing.

Another sad factor that contributes to confusion is that, alone of all the major .NET namespaces, the **System.IO** namespace violates good object-oriented design principles. In chapter 7, we spoke of the benefits of aggregating interfaces to specify the mix of abstract data types in an implementation. Rather than do that, the .NET IO classes have an overly inclusive **Stream** base class and a trio of public instance properties **CanRead**, **CanWrite**, and **CanSeek** that substitute for what should be type information. The motivation for this was probably a well-meaning desire to avoid an "explosion" in types and interfaces, but good design is as simple as possible *and no simpler*. By going too far with **Stream**, and with an

unfortunate handful of naming and behavior inconsistencies, the **System.IO** namespace can be quite frustrating.

Types of Stream

Classes descended from **Stream** come in two types: implementation classes associated with a particular type of data sink or source such as these three:

1. **MemoryStream**s are the simplest streams and work with in-memory data representations
2. **FileStream**s work with files and add functions for locking the file for exclusive access. **IsolatedStorageFileStream** descends from **FileStream** and is used by isolated stores.
3. **NetworkStream**s are very helpful when network programming and encapsulate (but provide access to) an underlying network socket. We are not going to discuss **NetworkStream**s until Chapter 18.

And classes which are used to dynamically add additional responsibilities to other streams such as these two:

1. **CryptoStream**s can encode and decode any other streams, whether those streams originate in memory, the file system, or over a network.
2. **BufferedStream**s improve the performance of most stream scenarios by reading and writing bytes in large chunks, rather than one at a time.

Classes such as **CryptoStream** and **BufferedStream** are called "Wrapper" or "Decorator" classes (see *Thinking in Patterns*).

Text and binary

Having determined where the stream is to exist (memory, file, or network) and how it is to be decorated (with cryptography and buffering), you'll need to choose whether you want to deal with the stream as characters or as bytes. If as characters, you can use the **StreamReader** and **StreamWriter** classes to deal with the data as lines of strings, if as bytes, you can use **BinaryReader** and **BinaryWriter** to translate bytes to and from the primitive value types.

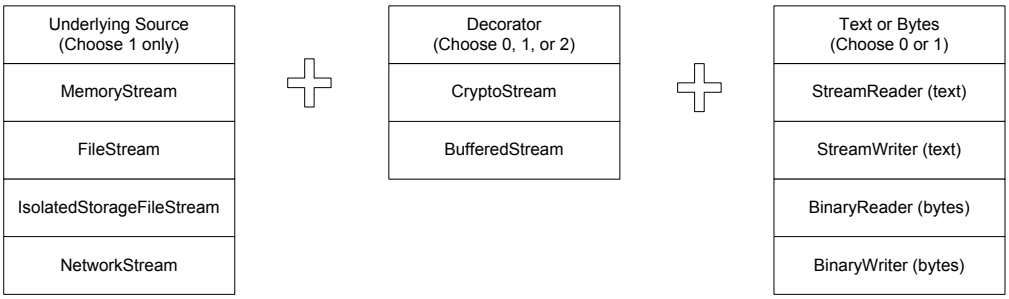


Figure 12-1: Streams use the Decorator pattern to build up capability

All told, there are 90 different valid combinations of these three aspects and while it can be confusing at first, it's clearer than having, for instance, a class called **BinaryCryptoFileReader**. Just to keep you on your toes, though, **StreamReader** and **StreamWriter** have sibling classes **StringReader** and **StringWriter** which work directly on strings, not streams.

Working with different sources

This example shows that although the way in which one turns a source into a stream differs, once in hand, any source can be treated equivalently:

```
//:c12:SourceStream.cs
using System;
using System.Text;
using System.IO;

class SourceStream {
  Stream src;

  SourceStream(Stream src){
    this.src = src;
  }

  void ReadAll(){
    Console.WriteLine(
      "Reading stream of type " + src.GetType());
    int nextByte;
    while ((nextByte = src.ReadByte()) != -1) {
      Console.Write((char) nextByte);
    }
  }
```

```
  public static void Main(){
    SourceStream srcStr = ForMemoryStream();
    srcStr.ReadAll();
    srcStr = ForFileStream();
    srcStr.ReadAll();
  }

  static SourceStream ForMemoryStream(){
    string aString = "mary had a little lamb";
    UnicodeEncoding ue = new UnicodeEncoding();
    byte[] bytes = ue.GetBytes(aString);
    MemoryStream memStream =
      new MemoryStream(bytes);
    SourceStream srcStream =
      new SourceStream(memStream);
    return srcStream;
  }

  static SourceStream ForFileStream(){
    string fName = "SourceStream.cs";
    FileStream fStream =
      new FileStream(fName, FileMode.Open);
    SourceStream srcStream =
      new SourceStream(fStream);
    return srcStream;
  }
}///:~
```

The constructor to **SourceStream** takes a **Stream** and assigns it to the instance variable **src**, while the method **ReadAll( )** reads that **src** one byte at a time until the method returns -1, indicating that there are no more bytes. Each byte read is cast to a char and sent to the console. The **Main( )** method uses the static methods **ForMemoryStream( )** and **ForFileStream( )** to instantiate **SourceStream**s and then calls the **ReadAll( )** method.

So far, all the code has dealt with **Stream**s no matter what their real source, but the static methods must necessarily be specific to the subtype of **Stream** being created. In the case of the **MemoryStream**, we start with a **string**, use the **UnicodeEncoding** class from the **System.Text** namespace to convert the **string** into an array of **bytes**, and pass the result into the **MemoryStream** constructor. The **MemoryStream** goes to the **SourceStream** constructor, and then we return the **SourceStream**.

For the **FileStream**, on the other hand, we have to specify an extant filename and what **FileMode** we wish to use to open it. The **FileMode** enumeration includes:

FileMode.Value	Behavior
Append	If the file exists, open it and go to the end of the file immediately. If the file does not exist, create it. File cannot be read.
Create	Creates a new file of the given name, *even if* that file already exists (it erases the extant file).
CreateNew	Creates a new file, *if it does not exist*. If the file exists, throws an **IOException**.
Open	Opens an existing file and throws a **FileNotFoundException** otherwise.
OpenOrCreate	Creates and opens a file, creating a new file if necessary
Truncate	Opens an existing file and truncates its size to zero.

Fun with CryptoStream**s**

Microsoft has done a big favor to eCommerce developers by including industrial-strength cryptography support in the .NET Framework. Many people mistakenly believe that to be anything but a passive consumer of cryptography requires hardcore mathematics. Not so. While few people are capable of developing new fundamental algorithms, cryptographic *protocols* that use the algorithms for complex tasks are accessible to anyone, while actual *applications* that use these protocols to deliver business value are few and far between.

Cryptographic algorithms come in two fundamental flavors: *symmetric algorithms* use the same key to encrypt and decrypt a data stream, while *asymmetric algorithms* have a "public" key for encryption and a "private" key for decryption. The .NET Framework comes with several symmetric algorithms and two asymmetric algorithms, one for general use and one that supports the standard for digital signatures.

Category	Name	Characteristics
Symmetric	DES	Older US Federal standard for "sensitive but not classified" data. 56-bit effective key. Cracked in 22 hours by $250,000 custom computer, plus 100K distributed PCs. If it's worth encrypting, it's worth not using DES.
Symmetric	TripleDES	An extension of DES that has a 112-bit effective key (note that this increases cracking difficulty by 2^{56}).
Symmetric	RC2	Variable key size, implementation seems to be fastest symmetric.
Symmetric	Rijndael	Algorithm chosen for Advanced Encryption Standard, effectively DES replacement. Fast, variable and large key sizes, generally the best symmetric cipher. Pronounced "rain-dahl"
Asymmetric	DSA	Cannot be used for encryption; only good for digital signing.
Asymmetric	RSA	Patent expired, almost synonymous with public-key cryptopgraphy.

CryptoStreams are only created by the symmetric algorithms.

```
//:c12:SecretCode.cs
using System;
using System.IO;
using System.Security.Cryptography;

class SecretCode {
  string fileName;
  string FileName{
    get { return fileName;}
    set { fileName = value;}
  }

  RijndaelManaged rm;
```

```
  SecretCode(string fName){
    fileName = fName;
    rm = new RijndaelManaged();
    rm.GenerateKey();
    rm.GenerateIV();
  }

  void EncodeToFile(string outName){
    FileStream src = new FileStream(
      fileName, FileMode.Open);

    ICryptoTransform encoder =
    rm.CreateEncryptor();
    CryptoStream str = new CryptoStream(
      src, encoder, CryptoStreamMode.Read);

    FileStream outFile = new FileStream(
      outName, FileMode.Create);

    int i = 0;
    while ((i = str.ReadByte()) != -1) {
      outFile.WriteByte((byte)i);
    }

    src.Close();
    outFile.Close();
  }

  void Decode(string cypherFile){
    FileStream src = new FileStream(
      cypherFile, FileMode.Open);

    ICryptoTransform decoder =
      rm.CreateDecryptor();
    CryptoStream str = new CryptoStream(
      src, decoder, CryptoStreamMode.Read);

    int i = 0;
    while ((i = str.ReadByte()) != -1) {
      Console.Write((char) i);
    }
```

```
    src.Close();
  }

  public static void Main(string[] args){
    SecretCode sc = new SecretCode(args[0]);
    sc.EncodeToFile("encoded.dat");
    Console.WriteLine("Decoded:");
    sc.Decode("encoded.dat");
  }
}///:~
```

The cryptographic providers are in the System.Security.Cryptography namespace. Each algorithm has both a base class named after the algorithm (DES, Rijndael, RSA, etc.) and an implementation of that algorithm provided by Microsoft. This is a nice design, allowing one to plug in new implementations of various algorithms as desired.

The System.Security.Cryptography namespace is not part of Microsoft's submission to ECMA and therefore the source code is not available to scrutiny as part of the shared-source Common Language Infrastructure initiative that Microsoft is trying to use to generate good will in the academic community. Although Microsoft's implementations have been validated by the US and Canadian Federal governments, it's a pity that this source code is not available for public review.

In this case, we use the RijndaelManaged class that implements the Rijndael algorithm. Like the other implementations, the RijndaelManaged class is able to generate random keys and initialization vectors, as shown in the **SecretCode** constructor, which also sets an instance variable **fileName** to the name of the file which we'll be encrypting.

EncodeToFile( ) opens a **FileStream** named **src** to our to-be-encrypted file. The symmetric cryptographic algorithms each provide a **CreateEncryptor( )** and **CreateDecryptor( )** method which returns an **ICryptoTransform** that is a necessary parameter for the **CryptoStream** constructor. With the input stream **src**, the **ICryptoTransform** encoder, and the **CryptoStreamMode.Read** mode as parameters we generate a **CryptoStream** called **str**.

The **outFile** stream is constructed in a familiar way but this time with **FileMode.Create**. We read the **str CryptoStream** and write it to the **outFile**, using the method **WriteByte( )**. Once done, we close both the source file and the newly created encrypted file.

The method **Decode( )** does the complement; it opens a **FileStream**, uses the **RijndaelManaged** instance to create an **ICryptoTransform decoder** and a **CryptoStream** appropriate for reading the encrypted file. We read the encrypted file one byte at a time and print the output on the console.

The **Main( )** method creates a new **SecretCode** class, passing in the first command-line argument as the filename to be encoded. Then, the call to **EncodeToFile( )** encrypts that file to another called "encoded.dat." Once that file is created, it is in turn decoded by the **Decode( )** method.

One characteristic of a good encrypted stream is that it is difficult to distinguish from a stream of random data; since random data is non-compressible, if you attempt to compress "encoded.dat" you should see that sure enough the "compressed" file is larger than the original.

BinaryReader **and** BinaryWriter

While we've been able to get by with reading and writing individual bytes, doing so requires a lot of extra effort when dealing with anything but the simplest data. **BinaryReader** and **BinaryWriter** are wrapper classes which can ease the task of dealing with the most common primitive value types. The BinaryWriter class contains a large number of overridden **Write( )** methods, as illustrated in this sample:

```
//:c12:BinaryWrite.cs
using System;
using System.IO;

class BinaryWrite {
  public static void Main(){
    Stream fStream = new FileStream(
      "binaryio.dat", FileMode.Create);
    WriteTypes(fStream);
    fStream.Close();
  }

  static void WriteTypes(Stream sink){
    BinaryWriter bw = new BinaryWriter(sink);

    bw.Write(true);
    bw.Write(false);
    bw.Write((byte) 7);
    bw.Write(new byte[]{ 1, 2, 3, 4});
```

```
    bw.Write('z');
    bw.Write(new char[]{ 'A', 'B', 'C', 'D'});
    bw.Write(new Decimal(123.45));
    bw.Write(123.45);
    bw.Write((short) 212);
    bw.Write((long) 212);
    bw.Write("<boolean>true</boolean>");
  }
}///:~
```

BinaryWrite's Main() method creates a FileStream for writing, upcasts the result to **Stream**, passes it to the static **WriteTypes( )** method, and afterwards closes it. The **WriteTypes( )** method takes the passed in **Stream** and passes it as a parameter to the **BinaryWriter** constructor. Then, we call **BinaryWriter.Write( )** with various parameters, everything from **bool** to **string**. Behind the scenes, the **BinaryWriter** turns these types into sequences of bytes and writes them to the underlying stream.

Every type, except for **string**, has a predetermined length in bytes – even bools, which could be represented in a single bit, are stored as a full byte — so it might be more accurate to call this type of storage "byte data" rather than "binary data." To store a string, **BinaryWriter** first writes one or more bytes to indicate the number of bytes that the string requires for storage; these bytes use 7 bits to encode the length and the 8th bit (if necessary) to indicate that the next byte is not the first character of the string, but another length byte.

The **BinaryWriter** class does nothing we couldn't do on our own, but it's much more convenient. Naturally, there's a complementary **BinaryReader** class, but because one cannot have polymorphism based only on return type (see chapter 8), the methods for reading various types are a little longer:

```
//:c12:BinaryRead.cs
using System;
using System.IO;

class BinaryRead {
  public static void Main(string[] args){
    Stream fStream = new BufferedStream(
      new FileStream(args[0], FileMode.Open));
    ByteDump(fStream);
    fStream.Close();
    fStream = new BufferedStream(
      new FileStream(args[0], FileMode.Open));
```

```
    ReadTypes(fStream);
    fStream.Close();
  }

  static void ByteDump(Stream src){
    int i = 0;
    while ((i = src.ReadByte()) != -1) {
      Console.WriteLine("{0} = {1} ", (char) i, i);
    }
    Console.WriteLine();
  }

  static void ReadTypes(Stream src){
    BinaryReader br = new BinaryReader(src);

    bool b = br.ReadBoolean();
    Console.WriteLine(b);
    b = br.ReadBoolean();
    Console.WriteLine(b);
    byte bt = br.ReadByte();
    Console.WriteLine(bt);
    byte[] byteArray = br.ReadBytes(4);
    Console.WriteLine(byteArray);
    char c = br.ReadChar();
    Console.WriteLine(c);
    char[] charArray = br.ReadChars(4);
    Console.WriteLine(charArray);
    Decimal d = br.ReadDecimal();
    Console.WriteLine(d);
    Double db = br.ReadDouble();
    Console.WriteLine(db);
    short s = br.ReadInt16();
    Console.WriteLine(s);
    long l = br.ReadInt64();
    Console.WriteLine(l);
    string tag = br.ReadString();
    Console.WriteLine(tag);
  }
}///:~
```

BinaryRead.Main( ) introduces another wrapper class, **BufferedStream**, which increases the efficiency of non-memory-based streams by using an internal

memory buffer to temporarily store the data rather than writing a single byte to the underlying file or network. BufferedStreams are largely transparent to use, although the method **Flush( )**, which sends the contents of the buffer to the underlying stream, regardless of whether it's full or not, can be used to fine-tune behavior.

BinaryRead works on a file whose name is passed in on the command line. **ByteDump( )** shows the contents of the file on the console, printing the byte as both a character and displaying its decimal value. When run on "binaryio.dat", the run begins:

```
☺ = 1
  = 0
 = 7
☺ = 1
☻ = 2
♥ = 3
♦ = 4
z = 122
A = 65
B = 66
C = 67
D = 68
…etc…
```

The first two bytes represent the Boolean values true and false, while the next parts of the file correspond directly to the values of the bytes and chars we wrote with the program **BinaryWrite**. The more complicated data types are harder to interpret, but towards the end of this method, you'll see a byte value of 212 that corresponds to the **short** and the **long** we wrote.

The last part of the output from this method looks like this:

```
↨ = 23
< = 60
b = 98
o = 111
o = 111
l = 108
e = 101
a = 97
n = 110
> = 62
```

```
t = 116
r = 114
u = 117
e = 101
< = 60
/ = 47
b = 98
o = 111
o = 111
l = 108
e = 101
a = 97
n = 110
> = 62
```

This particular string, which consumes 24 bytes of storage (1 length byte, and 23 character bytes), is the XML equivalent of the single byte at the beginning of the file that stores a **bool**. We'll discuss XML in length in chapter 17, but this shows the primary trade-off between binary data and XML – efficiency versus descriptiveness. Ironically, while local storage is experiencing greater-than-Moore's-Law increases in data density (and thereby becoming cheaper and cheaper) and network bandwidth (especially to the home and over wireless) will be a problem for the foreseeable future, file formats remain primarily binary and XML is exploding as the over-network format of choice!

After **BinaryRead** dumps the raw data to the console, it then reads the same stream, this time with the static method **ReadTypes( )**. **ReadTypes( )** instantiates a **BinaryReader( )** and calls its various **Readxxx( )** methods in exact correspondence to the **BinaryWriter.Write( )** methods of **BinaryWrite.WriteTypes( )**. When run on binaryio.dat, **BinaryRead.ReadTypes( )** reproduces the exact data, but you can also run the program on any file and it will gamely interpret that program's bytes as the specified types. Here's the output when **BinaryRead** is run on its own source file:

```
True
True
58
System.Byte[]
B
inar
-3.573226792213663651718845708 1E-75
```

```
6.2763486340252E-245
29962
8320773185183050099
em.IO;

class BinaryRead{
   public static void Main(string[] args){
      Stream fStream = new BufferedStream(
```

Again, this is the price to be paid for the efficiency of byte data – the slightest discrepancy between the types specified when the data is written and when it is read leads to incorrect data values, but the problem will probably not be detected until some *other* method attempts to use this wrong data.

StreamReader **and** StreamWriter

Because strings are such a common data type, the .NET Framework provides some decorator classes to aid in reading lines and blocks of text. The **StreamReader** and **StreamWriter** classes decorate streams and **StringReader** and **StringWriter** decorate **string**s.

The most useful method in **StreamReader** is **ReadLine( )**, as demonstrated in this sample, which prints a file to the console with line numbers prepended:

```
//:c12:LineAtATime.cs
using System;
using System.IO;

class LineAtATime {
  public static void Main(string[] args){
    foreach(string fName in args){
      Stream src = new BufferedStream(
        new FileStream(fName, FileMode.Open));
      LinePrint(src);
      src.Close();
    }
  }

  static void LinePrint(Stream src){
    StreamReader r = new StreamReader(src);
    int line = 0;
    string aLine = "";
    while ((aLine = r.ReadLine()) != null) {
```

```
      Console.WriteLine("{0}: {1}", line++, aLine);
    }
  }
}///:~
```

The **Main( )** method takes a command-line filename and opens it, decorates the **FileStream** with a **BufferedStream**, and passes the resulting **Stream** to the **LinePrint( )** static method. **LinePrint( )** creates a new **StreamReader** to decorate the **BufferedStream** and uses **StreamReader.ReadLine( )** to read the underlying stream a line at a time. **StreamReader.ReadLine( )** returns a null reference at the end of the file, ending the output loop.

StreamReader is useful for writing lines and blocks of text, and contains a slew of overloaded **Write( )** methods similar to those in BinaryWriter, as this example shows:

```
//:c12:TextWrite.cs
using System;
using System.IO;

class TextWrite {
  public static void Main(){
    Stream fStream = new FileStream(
      "textio.dat", FileMode.Create);
    WriteLineTypes(fStream);
    fStream.Close();
  }

  static void WriteLineTypes(Stream sink){
    StreamWriter sw = new StreamWriter(sink);

    sw.WriteLine(true);
    sw.WriteLine(false);
    sw.WriteLine((byte) 7);
    sw.WriteLine('z');
    sw.WriteLine(new char[]{ 'A', 'B', 'C', 'D'});
    sw.WriteLine(new Decimal(123.45));
    sw.WriteLine(123.45);
    sw.WriteLine((short) 212);
    sw.WriteLine((long) 212);
    sw.WriteLine("{0} : {1}",
      "string formatting supported", "true");
```

```
    sw.Close();
  }
}///:~
```

Like the BinaryWrite sample, this program creates a filestream (this time for a file called "textio.dat") and passes that to another method that decorates the underlying data sink and writes to it. In addition to **Write( )** methods that are overloaded to write the primitive types, **StreamWriter** will call **ToString( )** on *any* object and supports string formatting. In one of the namespace's annoyances, **StreamWriter** is buffered (although it doesn't descend from **BufferedStream**), and so you must explicitly call **Close( )** in order to flush the lines to the underlying stream.

The data written by **StreamWriter** is in text format, as shown in the contents of textio.dat:

```
True
False
7
z
ABCD
123.45
123.45
212
212
string formatting supported : true
```

Bear in mind that **StreamReader** does not have **Readxxx( )** methods – if you want to store primitive types to be read and used as primitive types, you should use the byte-oriented **Reader** and **Writer** classes. You *could* store the data as text, read it as text, and then perform the various string parsing operations to recreate the values, but that would be wasteful.

It's worth noting that **StreamReader** and **StreamWriter** have sibling classes **StringReader** and **StringWriter** that are descended from the same **Reader** and **Writer** abstraction. Since **string** objects are immutable (once set, a **string** cannot be changed), there is a need for an efficient tool for building strings and complex formatting tasks. The basic task of building a string from substrings is handled by the **StringBuilder** class, while the complex formatting can be done with the **StringWriter** (which decorates a **StringBuilder** in the same way that the **StreamWriter** decorates a **Stream**).

Random access with Seek

The Stream base class contains a method called **Seek( )** that can be used to jump between records and data sections of known size (or sizes that can be computed by reading header data in the stream). The records don't have to be the same size; you just have to be able to determine how big they are and where they are placed in the file. The Seek() method takes a long (implying a maximum file size of 8 exabytes, which will hopefully suffice for a few years) and a value from the **SeekOrigin** enumeration which can be **Begin**, **Current**, or **End**. The SeekOrigin value specifies the point from which the seek jumps.

Although **Seek( )** is defined in **Stream**, not all **Stream**s support it (for instance, one can't "jump around" a network stream). The **CanSeek** bool property specifies whether the stream supports **Seek( )** and the related **Length( )** and **SetLength( )** mehods, as well as the **Position( )** method which returns the current position in the **Stream**. If **CanSeek** is **false** and one of these methods is called, it will throw a **NotSupportedException**. This is poor design. Support for random access is based on type, not state, and should be specified in an interface (say, **ISeekable**) that is implemented by the appropriate subtypes of **Stream**.

If you use **SeekOrigin.End**, you should use a negative number for the offset; performing a **Seek( )** beyond the end of the **stream** moves to the end of the file (i.e., **ReadByte( )** will return a -1, etc.).

This example shows the basic use of **Stream.Seek( )**:

```
//:c12:FibSeek.cs
using System;
using System.IO;

class FibSeek {
  Stream src;

  FibSeek(Stream src){
    this.src = src;
  }

  void DoSeek(SeekOrigin so){
    if (so == SeekOrigin.End) {
      src.Seek(-10, so);
    } else {
```

```
      src.Seek(10, so);
    }
    int i = src.ReadByte();
    Console.WriteLine(
      "10 bytes from {0} is : {1}", so, (char) i);
  }

  public static void Main(string[] args){
    foreach(string fName in args){
      FileStream f = null;
      try {
        f = new FileStream(fName, FileMode.Open);
        FibSeek fs = new FibSeek(f);
        fs.DoSeek(SeekOrigin.Begin);
        fs.DoSeek(SeekOrigin.End);
        f.Seek(12, SeekOrigin.Begin);
        fs.DoSeek(SeekOrigin.Current);
      } catch (Exception ex) {
        Console.WriteLine(ex);
      } finally {
        f.Close();
      }
    }
  }
}///:~
```

Standard I/O

The term *standard I/O* refers to the Unix concept (which is reproduced in some form in Windows and many other operating systems) of a single stream of information that is used by a program. All the program's input can come from *standard input*, all its output can go to *standard output*, and all of its error messages can be sent to *standard error*. The value of standard I/O is that programs can easily be chained together and one program's standard output can become the standard input for another program. More than just a convenience, this is a powerful architectural pattern called *Pipes and Filters*; although this architecture was not very common in the 1990s, it's a very powerful one, as anyone who's witnessed a UNIX guru can testify.

Reading from standard input

Following the standard I/O model, the **Console** class exposes three static properties: **Out**, **Error**, and **In**. In Chapter 11 we sent some error messages to **Console.Error**. **Out** and **Error** are **TextWriter**s, while **In** is a **TextReader**.

Typically, you either want to read console input as either a character or a complete line at a time. Here's an example that simply echoes each line that you type in:

```
//:c12:EchoIn.cs
//How to read from standard input.
using System;

public class EchoIn {
  public static void Main(){
    string s;
    while ((s = Console.In.ReadLine()).Length != 0)
      Console.WriteLine(s);
    // An empty line terminates the program
  }
} ///:~
```

Redirecting standard I/O

The **Console** class allows you to redirect the standard input, output, and error I/O streams using simple static method calls:

SetIn(TextReader)
SetOut(TextWriter)
SetError(TextWriter)

(There is no obvious reason why these methods are used rather than allowing the Properties to be set directly.)

Redirecting output is especially useful if you suddenly start creating a large amount of output on your screen and it's scrolling past faster than you can read it. Redirecting input is valuable for a command-line program in which you want to test a particular user-input sequence repeatedly. Here's a simple example that shows the use of these methods:

```
//:c12:Redirecting.cs
// Demonstrates standard I/O redirection.
using System;
```

```
using System.IO;

public class Redirecting {
  public static void Main(){
    StreamReader sr = new StreamReader(
      new BufferedStream(
        new FileStream(
          "Redirecting.cs", FileMode.Open)));
    StreamWriter sw = new StreamWriter(
      new BufferedStream(
        new FileStream(
          "redirect.dat", FileMode.Create)));
    Console.SetIn(sr);
    Console.SetOut(sw);
    Console.SetError(sw);

    String s;
    while ((s = Console.In.ReadLine()) != null)
      Console.Out.WriteLine(s);
    Console.Out.Close(); // Remember this!
  }
} ///:~
```

This program attaches standard input to a file, and redirects standard output and standard error to another file.

Debugging and Tracing

We briefly discussed the **Debug** and **Trace** classes of the **System.Diagnostics** namespace in chapter 6. These classes are enabled by conditionally defining the values **DEBUG** and **TRACE** either at the command-line or in code. These classes write their output to a set of **TraceListener** classes. The default **TraceListener** of the **Debug** class interacts with the active debugger, that of the **Trace** class sends data to the console. Customizing both is easy; the **TextWriterTestListener** decorates any **TextWriter** with **TestListener** capabilities. Additionally, **EventLogTraceListener** ; sending data to the console or the system's event logs takes just a few lines of code:

```
//:c12:DebugAndTrace.cs
//Demonstates Debug and Trace classes
#define DEBUG
#define TRACE
```

```
using System;
using System.Diagnostics;

class DebugAndTrace {
  public static void Main(){
    TextWriterTraceListener conWriter =
    new TextWriterTraceListener(Console.Out);
    Debug.Listeners.Add(conWriter);
    Debug.WriteLine("Debug to stdout");

    EventLogTraceListener logWriter =
    new EventLogTraceListener("DebugTraceProg");
    Trace.Listeners.Add(logWriter);
    Debug.Listeners.Add(logWriter);

    Trace.WriteLine("Traced");
    Debug.WriteLine("Debug trace");
    logWriter.Close();
  }
}///:~
```

When run, both **Debug** and **Trace** are written to the console. In addition, an **EventLogTraceListener** object whose **Source** property is set to "DebugTraceLog." This value is used to show in the system's event logs the source of trace information:

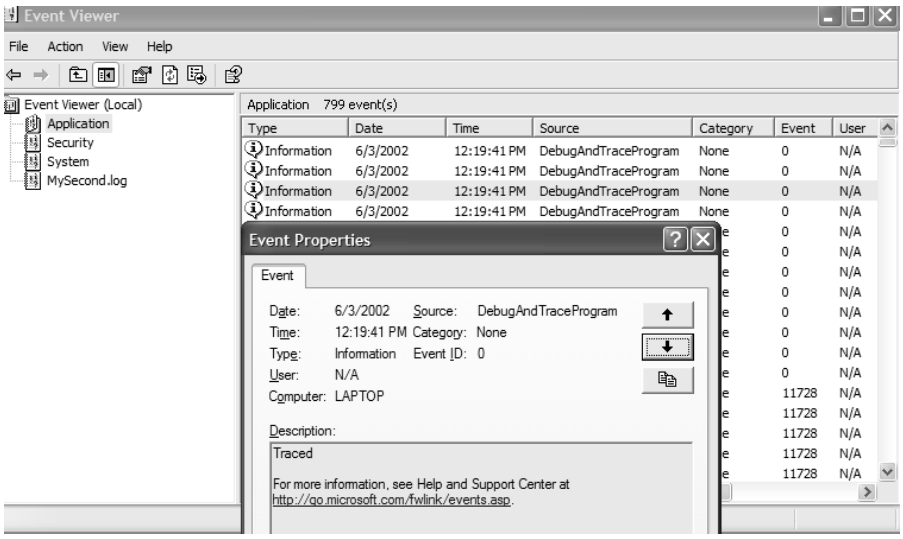


Figure 12-2: Using the system Event Viewer to see program output

If you wish to create your own event log, that's easy, too:

```
EventLog log = new EventLog("MySecond.log");
log.Source = "DebugAndTraceProgram";
EventLogTraceListener logWriter =
   new EventLogTraceListener(log);
```

I think this section could be expanded a bit.

Regular expressions

Regular expressions are a powerful pattern-matching tool for interpreting and manipulating strings. Although regular expressions are not necessarily related to input and output, it is probably their most common application, so we'll discuss them here.

Regular expressions have a long history in the field of computer science but continue to be expanded and improved, which gives rise to an intimidating set of capabilities and alternate routes to a given end. The regular expressions in the .NET Framework are Perl 5 compatible but include additional features such as right-to-left matching and do not require a separate compilation step.

The fundamental responsibility of the **System.Text.RegularExpressions Regex** class is to match a given pattern with a given target string. The pattern is described in a terse notation that combines literal text that must appear in the

target with meta-text that specifies both acceptable variations in text and desired manipulations such as variable assignment or text replacement.

This sample prints out the file names and lines that match a regular expression typed in the command line:

```
//:c12:TGrep.cs
//Demonstrate basic regex matching against files
using System;
using System.IO;
using System.Text.RegularExpressions;

class TGrep {
  public static void Main(string[] args){
    TGrep tg = new TGrep(args[0]);
    tg.ApplyToFiles(args[1]);
  }
  Regex re;

  TGrep(string pattern){
    re = new Regex(pattern);
  }

  void ApplyToFiles(string fPattern){
    string[] fNames =
    Directory.GetFiles(".", fPattern);
    foreach (string fName in fNames ) {
      StreamReader sr = null;
      try {
        sr = new StreamReader(
          new BufferedStream(
            new FileStream(
              fName, FileMode.Open)));
        string line = "";
        int lCount = 0;
        while ((line = sr.ReadLine()) != null) {
          lCount++;
          if (re.IsMatch(line)) {
            Console.WriteLine(
              "{0} {1}: {2}", fName, lCount, line);
          }
        }
```

```
      } finally {
        sr.Close();
      }
    }
  }
}///:~
```

The **Main( )** method passes the first command-line argument to the **TGrep( )** constructor, which in turn passes it to the **Regex( )** constructor. The second argument is then passed as the argument to the **ApplyToFiles( )** method.

ApplyToFiles( ) uses IO techniques we've discussed previously to read a series of files line-by-line and incrementing the variable **lCount** to let us know what line number works. Each line is passed to the **Regex.IsMatch( )** method, and if that method returns **true**, the filename, line number, and contents of the line are printed to the screen.

You might guess that "tgrep using tgrep.cs" would print lines 3, 4, and 5 of **tgrep.cs**, but you might not expect that "tgrep [0-9] tgrep.cs" would print every line that contains a number, or that "tgrep [\s]f[\w]*[\s]*= *.cs" would print every line that assigns a value to a variable that begins with a lowercase "f". Like SQL in ADO.NET, the regular expression notation is a separate language quite unlike C#, and *Thinking in Regular Expressions* would be quite a different book than this one.

In addition to simply determining if a match exists, **Regex** can actually return the value of the matches, as this program demonstrates:

```
//:c12:GrepMatches.cs
using System;
using System.IO;
using System.Text.RegularExpressions;

class GrepMatches {
  public static void Main(string[] args){
    GrepMatches tg = new GrepMatches(args[0]);
    string target = args[1];
    tg.ApplyToFiles(target);
  }
  Regex re;

  GrepMatches(string pattern){
    re = new Regex(pattern);
```

```
  }

  void ApplyToFiles(string fPattern){
    string[] fNames = Directory.GetFiles(
      ".", fPattern);
    foreach (string fName in fNames ) {
      StreamReader sr = null;
      try {
        sr = new StreamReader(
          new BufferedStream(
            new FileStream(fName, FileMode.Open)));
        string line = "";
        int lCount = 0;
        while ((line = sr.ReadLine()) != null) {
          lCount++;
          if (re.IsMatch(line)) {
            Console.WriteLine(
              "{0} {1}: {2}", fName, lCount, line);
            ShowMatches(re.Matches(line));
          }
        }
      } finally {
        sr.Close();
      }
    }
  }

  private void ShowMatches(MatchCollection mc){
    for (int i = 0; i < mc.Count; i++) {
      Console.WriteLine(
        "Match[{0}] = {1}", i, mc[i]);
    }
  }
}///:~
```

Regex.Matches( ) returns a **MatchCollection** which naturally contains **Match** objects. This sample program can be helpful in debugging the development of a regular expression, which for most of us requires a considerable amount of trial and error!

The static method Regex.Replace() can make complex transformations surprisingly straightforward. This sample makes pattern substitutions in a text file:

```
//:c12:TSed.cs
using System;
using System.IO;
using System.Text.RegularExpressions;

class TSed {
  public static void Main(string[] args){
    TSed tg = new TSed(args[0], args[1]);
    string target = args[2];
    tg.ApplyToFiles(target);
  }

  string pattern;
  string rep;

  TSed(string pattern, string rep){
    this.pattern = pattern;
    this.rep = rep;
  }

  void ApplyToFiles(string fPattern){
    string[] fNames =
      Directory.GetFiles(".", fPattern);
    foreach (string fName in fNames ) {
      StreamReader sr = null;
      try {
        sr = new StreamReader(
          new BufferedStream(
            new FileStream(fName, FileMode.Open)));
        string line = "";
        int lCount = 0;
        while ((line = sr.ReadLine()) != null) {
          string nLine =
          Regex.Replace(line, pattern, rep);
          Console.WriteLine(nLine);
        }
      } finally {
```

```
        sr.Close();
      }
    }
  }
}///:~
```

Like the previous samples, this one works with command-line arguments, but this time, instead of instantiating a **Regex** for pattern-matching, the first two command-line arguments are just stored as strings, which are later passed to the **Regex.Replace( )** method. If the pattern matches, the replacement pattern is inserted into the string, if not, the line is untouched. Whether touched or not, the line is written to the console; this makes this program a "tiny" version of UNIX's sed command and is very convenient.

Checking capitalization style

In this section we'll look at a complete example of the use of C# IO which also uses regular expression. This project is directly useful because it performs a style check to make sure that your capitalization conforms to the C# style. It opens each **.cs** file in the current directory and extracts all the class names and identifiers, then shows you if any of them don't meet the C# style. You can then use the TSed sample above to automatically replace them.

The program uses two regular expressions that match words that precede a block and which begin with a lowercase letter. One **Regex** matches block-oriented identifiers (such as class, interface, property, and namespace names) and the other catches method declarations. Doing this in a single **Regex** is one of the exercises at the end of the chapter.

```
//:c12:CapStyle.cs
//Scans all .cs files for properly capitalized
//method and classnames
using System;
using System.IO;
using System.Text.RegularExpressions;

public class CapStyle {
  public static void Main(){
    string[] fNames =
      Directory.GetFiles(".","*.cs");
    foreach(string fName in fNames){
      CapStyle cs = null;
      try {
```

```
      cs = new CapStyle(fName);
      cs.Check();
    } finally {
      cs.Close();
    }
  }
}

string[] keyWords= new string[]{
  "abstract", "event", "new", "struct", "as",
  "explicit", "null", "switch", "base", "extern",
  "object", "this", "bool", "false", "operator",
  "throw", "break", "finally", "out", "true",
  "byte", "fixed", "override", "try", "case",
  "float", "params", "typeof", "catch", "for",
  "private", "uint", "char", "foreach",
  "protected", "ulong", "checked", "goto",
  "public", "unchecked", "class", "if",
  "readonly", "unsafe", "const", "implicit",
  "ref", "ushort", "continue", "in", "return",
  "using", "decimal", "int", "sbyte", "virtual",
  "default", "interface", "sealed", "volatile",
  "delegate", "internal", "short", "void", "do",
  "is", "sizeof", "while", "double", "lock",
  "stackalloc", "else", "long", "static", "enum",
  "namespace", "string", "try", "catch",
  "finally", "using", "else", "switch", "public",
  "static", "void", "foreach", "if", "while",
  "bool", "byte", "for", "get", "set"
};

StreamReader fStream;

Regex blockPrefix;
Regex methodDef;

CapStyle(string fName){
  fStream = new StreamReader(
    new BufferedStream(
      new FileStream(fName, FileMode.Open)));
  /*
```

```
    matches just-before-bracket identifier
    starting with lowercase
    */
    blockPrefix =
      new Regex(@"[\s](?<id>[a-z][\w]*)[\s]*{");

    /*
    matches just-before-bracket with argument list
    and identifier starting with lowerCase
    */
    methodDef =
      new Regex(
        @"[\s](?<id>[a-z][\w]*)\s*\((.*)\)[\s]*{");

    Console.WriteLine(
      "Checking file: " + fName);
  }

  void Close(){
    fStream.Close();
  }

  void Check(){
    string line = "";
    int lCount = 0;
    while ((line = fStream.ReadLine()) != null) {
      lCount++;
      if (Suspicious(line)) {
        Console.WriteLine(
          "{0}: {1}", lCount, line);
      }
    }
  }

  bool Suspicious(string line){
    if (MatchNotKeyword(line, blockPrefix) == true) {
      return true;
    }
    if (MatchNotKeyword(line, methodDef) == true) {
      return true;
    }
```

```
    return false;
  }

  bool MatchNotKeyword(string line, Regex re){
    if (re.IsMatch(line)) {
      Match m = re.Match(line);
      string identifier = m.Groups["id"].Value;
      if (Array.IndexOf(keyWords, identifier) < 0) {
        return true;
      }
    }
    return false;
  }
}///:~
```

The **Main( )** generates a list of all the C# files in the current directory and for each one creates a **CapStyle** instance, runs **CapStyle.Check( )** and then **CapStyle.Close( ).**

Each **CapStyle** instance contains a list of C# keywords that are allowed to be in lowercase, as well as instance variables that hold the two regular expressions, and a **StreamReader** instance variable that reads the underlying file. The **CapStyle( )** constructor opens the file and constructs the two two regular expressions. The expressions will match namespaces, class and interface identifiers, properties, and method names that precede a '{' character (handling multiline bracketing conventions is another exercise!). Additionally, the expressions use *group naming* to associate the word that begins with a lowercase letter to a regex variable called **id** ("`(?<id>[a-z][\w]*)`" is the relevant notation; the parentheses specify the group, the `?<id>` specifies the name).

The **Check( )** method goes through the **StreamReader** line-by-line, seeing if **Suspicious( )** returns true; if so, that line is output to the console. **Suspicious( )** in turn calls **MatchNotKeyword( )**, passing in the suspect line and a reference to one of the two instance **Regex**s. **MatchNotKeyword( )** checks for a match; if there is one it assigns the value of the **Regex** group named **id** to the string **identifier**. If this string does not appear in the array of C# keywords, **MatchNotKeyword( )** returns **true**, which causes **Suspicious** to return **true** to **Check( )**.

In addition to not handling multiline bracketing, this program sometimes marks **string**s that contain formatting brackets incorrectly. If you improve the program, please drop the authors a line at *www.ThinkingIn.Net*.

Summary

The .NET IO stream library does satisfy the basic requirements: you can perform reading and writing with the console, a file, a block of memory, or even across the Internet (as you will see in Chapter 18). With inheritance, you can create new types of input and output objects.

The IO library brings up mixed feelings; it does the job and it uses the Decorator pattern to good effect. But if you don't already understand the Decorator pattern, the design is nonintuitive, so there's extra overhead in learning and teaching it. There are also some poor choices in naming and implementation issues.

However, once you *do* understand the fundamentals of **Stream**s and the Decorator pattern and begin using the library in situations that require its flexibility, you can begin to benefit from this design, at which point its cost in extra lines of code will not bother you at all.

Exercises

1. Open a text file so that you can read the file one line at a time. Read each line as a **string** and place that **string** object into a **SortedList**. Print all of the lines in the **SortedList** in reverse order.
2. Modify the previous exercise so that the name of the file you read is provided as a command-line argument.
3. Modify the previous exercise to also open a text file so you can write text into it. Write the lines in the **SortedList**, along with line numbers, out to the file.
4. Modify Exerise 2 to force all the lines in the SortedList to upper case and send the results to the console.
5. Modify Exercise 2 to take additional command-line arguments of words to find in the file. Print all lines in which any of the words match.
6. Modify **DirList.cs** to actually open each file and only list those files whose contents contain any of the words specified on the command-line.
7. Modify **WordCount.cs** so that it produces an alphabetic sort.
8. Write a program that compares the performance of writing to a file when using buffered and unbuffered I/O.

9. Write a program that changes operators within a C# source code file (for instance, that changes addition operators into subtraction, or flips binary tests from **true** to **false**). Use this program to explore *mutation testing*, which starts from the premise that every operator ought to affect the behavior of the program.

10. Write a program that creates Markov chains. First, write a program that reads each word in a series of files and stores, for each word, the words that follow it and the probability of that word being next (for instance, the word ".Net" is likely to be followed by the words "framework" or "platform" more often than being followed by the word "crepuscular"). Once this data structure is created from a large enough corpus, generate new sentences by picking a common word, choosing a successor probabilistically (use **Random.NextDouble( )** and the fact that all probabilities sum to 1). Run the program on different source texts (press releases, Hemingway short stories, books on computer programming).

11. Incorporate punctuation, sentence length, and Markov chains longer than a single word into the previous example.

13: Reflection and Attributes

The idea of run-time type identification (RTTI) seems fairly simple at first: It lets you find the exact type of an object when you only have a reference to the base type.

However, the *need* for RTTI uncovers a whole plethora of interesting (and often perplexing) OO design issues, and raises fundamental questions of how you should structure your programs.

This chapter looks at the ways that C# allows you to add and discover information about objects and classes at run-time. This takes three forms: "traditional" RTTI, which assumes that you have all the types available at compile-time and run-time, the "reflection" mechanism, which allows you to discover class information solely at run-time, and the "attributes" mechanism, which allows you to declare new types of "meta-information" with a program element and write programs that recognize and work with that new meta-information. We'll cover these three mechanisms in order.

The need for RTTI

Consider the now familiar example of a class hierarchy that uses polymorphism. The generic type is the base class **Shape**, and the specific derived types are **Circle**, **Square**, and **Triangle**:

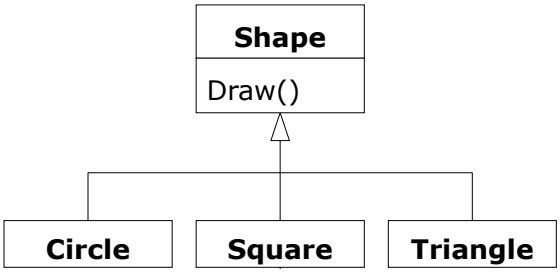


Figure 13-1: The Shape hierarchy

This is a typical class hierarchy diagram, with the base class at the top and the derived classes growing downward. The normal goal in object-oriented programming is for the bulk of your code to manipulate references to the base type (**Shape**, in this case), so if you decide to extend the program by adding a new class (**Rhomboid**, derived from **Shape**, for example), the bulk of the code is not affected. In this example, the dynamically bound method in the **Shape** interface is **Draw( )**, so the intent is for the client programmer to call **Draw( )** through a generic **Shape** reference. **Draw( )** is overridden in all of the derived classes, and because it is a dynamically bound method, the proper behavior will occur even though it is called through a generic **Shape** reference. That's polymorphism.

Thus, you generally create a specific object (**Circle**, **Square**, or **Triangle**), upcast it to a **Shape** (forgetting the specific type of the object), and use that **Shape** abstract data type reference in the rest of the program.

As a brief review of polymorphism and upcasting, you might code the above example as follows:

```
//:c13:Shapes.cs
using System;
using System.Collections;

class Shape {
  internal void Draw() {
    Console.WriteLine(this + ".Draw()");
  }
}

class Circle : Shape {
```

```
  public override string ToString() {
    return "Circle";}
}

class Square : Shape {
  public override string ToString() {
    return "Square";}
}

class Triangle : Shape {
  public override string ToString() {
    return "Triangle";}
}

public class Shapes {
  public static void Main() {
    IList s = new ArrayList();
    s.Add(new Circle());
    s.Add(new Square());
    s.Add(new Triangle());
    IEnumerator e = s.GetEnumerator();
    while (e.MoveNext()) {
      ((Shape)e.Current).Draw();
    }
  }
} ///:~
```

The base class contains a **Draw( )** method that indirectly uses **ToString( )** to print an identifier for the class by passing **this** to **Console.WriteLine( )**. If that function sees an object, it automatically calls the **ToString( )** method to produce a **String** representation.

Each of the derived classes overrides the **ToString( )** method (from **object**) so that **Draw( )** ends up printing something different in each case. In **Main( )**, specific types of **Shape** are created and then added to an **IList**. This is the point at which the upcast occurs because the **IList** holds only **object**s. Since everything in C# is an **object**, an **IList** can also hold **Shape** objects. But during an upcast to **object**, it also loses any specific information, including the fact that the objects are **Shape**s. To the **ArrayList**, they are just **object**s.

At the point you fetch an element out of the **IList**'s **IEnumerator** with **MoveNext( )**, things get a little busy. Since the **IList** holds only **object**s,

MoveNext( ) naturally produces an **object** reference. But we know it's really a **Shape** reference, and we want to send **Shape** messages to that object. So a cast to **Shape** is necessary using the traditional "**(Shape)**" cast. This is the most basic form of RTTI, since in C# all casts are checked at run-time for correctness. That's exactly what RTTI means: At run-time, the type of an object is identified.

In this case, the RTTI cast is only partial: The **object** is cast to a **Shape**, and not all the way to a **Circle**, **Square**, or **Triangle**. That's because the only thing we *know* at this point is that the **IList** is full of **Shape**s. At compile-time, this is enforced only by your own self-imposed rules, but at run-time the cast ensures it.

Now polymorphism takes over and the exact method that's called for the **Shape** is determined by whether the reference is for a **Circle**, **Square**, or **Triangle**. And in general, this is how it should be; you want the bulk of your code to know as little as possible about *specific* types of objects, and to just deal with the abstract data type that represents a family of objects (in this case, **Shape**). As a result, your code will be easier to write, read, and maintain, and your designs will be easier to implement, understand, and change. So polymorphism is the general goal in object-oriented programming.

But what if you have a special programming problem that's easiest to solve if you know the exact type of a generic reference? For example, suppose you want to allow your users to highlight all the shapes of any particular type by turning them purple. This way, they can find all the triangles on the screen by highlighting them. Or perhaps you have an external method that needs to "rotate" a list of shapes, but it makes no sense to rotate a circle so you'd like to skip only the circle objects. This is what RTTI accomplishes: you can ask a **Shape** reference the exact type that it's referring to. With RTTI you can select and isolate special cases.

The Type object

To understand how RTTI works in C#, you must first know how type information is represented at run-time. This is accomplished through a special kind of object called the *Type object,* which contains information about the class. (This is sometimes called a *meta-class.*) In fact, the **Type** object is used to create all of the "regular" objects of your class[1].

There's a **Type** object for each type that is part of your program. That is, each time you write and compile a new type, whether it be a value type such as a structure, or a "real" object, a single **Type** object is created. A collection of **Type**

[1] In fact, all your objects will be of type **RuntimeType**, which is a subtype of **Type**.

objects is stored in binary format in an assembly (usually having an extension of **.dll** or **.exe**). At run-time, when you want to make an object of that type, the CLR first checks to see if the **Type** has been instantiated within the current **AppDomain** (roughly, an **AppDomain** is the runtime container for the assemblies of a single application). If the type has not been instantiated, the CLR reads the assembly and transforms the CIL contents into machine instructions appropriate to the local hardware (this process is called *Just In Time Compilation*, and *JIT* has become a common verb to describe it*)*. This happens in every **AppDomain** that uses the **Type**; some amount of memory efficiency is traded for the benefits, such as security, that come from isolating **AppDomain**s. Thus, a .NET program isn't completely loaded before it begins, which is different from many traditional languages.

Once the **Type** object for that type is in memory, it is used to create all instances of that type.

If this seems shadowy or if you don't really believe it, here's a demonstration program to prove it:

```
//:c13:SweetShop.cs
// Examination of the way type loading works.
using System;

class Candy {
  static Candy(){
    Console.WriteLine("Candy loaded");
  }
}

class Gum {
  static Gum(){
    Console.WriteLine("Gum loaded");
  }

  internal static string flavor = "juicyfruit";
}

class Cookie {
  static Cookie() {
    Console.WriteLine("Cookie loaded");
  }
}
```

```
public class SweetShop {
  public static void Main() {
    Console.WriteLine("Inside Main");
    new Candy();
    Console.WriteLine("After creating Candy");
    Type t = Type.GetType("Gum");
    Console.WriteLine(
      "After Type.GetType(\"Gum\")");
    Console.WriteLine(Gum.flavor);
    Console.WriteLine("Before creating Cookie");
    new Cookie();
    Console.WriteLine("After creating Cookie");
  }
} ///:~
```

Each of the classes **Candy**, **Gum**, and **Cookie** have a **static** constructor that is executed the first time an instance of the class is created. Information will be printed to tell you when that occurs. In **Main( )**, the object creations are spread out between print statements to help detect the time of loading.

A particularly interesting sequence is:

```
Type t = Type.GetType("Gum");
Console.WriteLine(
      "After Type.GetType(\"Gum\")");
Console.WriteLine(Gum.flavor);
```

Type.GetType( ) is a static method that attempts to load a type of the given name. A **Type** object is like any other object and so you can get and manipulate a reference to it. One of the ways to get a reference to the **Type** object is **Type.GetType( )**, which takes a **string** containing the textual name of the particular class you want a reference for.

When you run this program, the output will be:

```
Inside Main
Candy loaded
After creating Candy
After Type.GetType("Gum")
Gum loaded
juicyfruit
```

```
Before creating Cookie
Cookie loaded
After creating Cookie
```

You can see that each Class object is loaded only when it's needed, and the **static** constructor is run immediately prior to when data from the **Type** is needed (in this case, the **static string** that told the **Gum**'s flavor). This is in slight contrast to Java, which instantiates the static state of a type immediately upon class loading.

Type retrieval operator

C# provides a second way to produce the reference to the **Type** object, using the *type retrieval operator* **typeof( )**. In the above program this would look like:

```
typeof(Gum);
```

which is not only simpler, but also safer since it's checked at compile-time. Because it eliminates the method call, it's also more efficient.

Checking before a cast

So far, you've seen RTTI forms including:

- The classic cast; e.g., "**(Shape),**" which uses RTTI to make sure the cast is correct.
- The **Type** object representing the type of your object. The **Type** object can be queried for useful run-time information.

In C++, the classic cast "**(Shape)**" does *not* perform RTTI. It simply tells the compiler to treat the object as the new type. In C#, which does perform the type check, this cast is often called a "type safe downcast." The reason for the term "downcast" is the historical arrangement of the class hierarchy diagram. If casting a **Circle** to a **Shape** is an upcast, then casting a **Shape** to a **Circle** is a downcast. However, you know a **Circle** is also a **Shape**, and the compiler freely allows an upcast assignment, but you *don't* know that a **Shape** is necessarily a **Circle**, so the compiler doesn't allow you to perform a downcast assignment without using an explicit cast.

There's one more form of RTTI in C#. These are the keyword **is** and **as**. The keyword **is** tells you if an object is an instance of a particular type. It returns a **bool** so you use it in the form of a question, like this:

```
if(cheyenne is Dog)
  ((Dog)cheyenne).Bark();
```

The above **if** statement checks to see if the object **cheyenne** belongs to the class **Dog** *before* casting **cheyenne** to a **Dog**. It's important to use **is** before a downcast when you don't have other information that tells you the type of the object; otherwise you'll end up with an **InvalidCastException**.

The keyword **as** performs a downcast to the specified type, but returns **null** if the object is not an object of the specified type. So the above example becomes:

```
Dog d = cheyenne as Dog;
d.Bark();
```

If the object **cheyenne** did not belong to class **Dog**, this would still compile fine, but you would get a **NullReferenceException** when the line **d.Bark( )** attempts to execute. You should exercise extreme caution with **as**, especially if you do not immediately attempt to use the result. Leaving possibly **null** values floating around is sloppy programming.

Ordinarily, you will be hunting for one type (triangles to turn purple, for example), but you can easily tally *all* of the objects using **is**. Suppose you have a family of **Pet** classes:

```
//:c13:Pets.cs
class Pet { }
class Dog :Pet { }
class Pug :Dog { }
class Cat :Pet { }
class Rodent :Pet { }
class Gerbil :Rodent { }
class Hamster :Rodent { }
///:~
```

Using **is**, all the pets can be counted:

```
//:c13:PetCount1.cs
//Compile with:
//csc Pets.cs PetCount1.cs
using System;
using System.Collections;

public class PetCount {
  static string[] typenames = {
    "Pet", "Dog", "Pug", "Cat",
    "Rodent", "Gerbil", "Hamster",
  };
```

```
public static void Main() {
  ArrayList pets = new ArrayList();
  Type[] petTypes = {
    Type.GetType("Dog"),
    Type.GetType("Pug"),
    Type.GetType("Cat"),
    Type.GetType("Rodent"),
    Type.GetType("Gerbil"),
    Type.GetType("Hamster"),
  };

  Random r = new Random();

  for (int i = 0; i < 15; i++) {
    Type t = petTypes[r.Next(petTypes.Length)];
    object o = Activator.CreateInstance(t);
    pets.Add(o);
  }

  Hashtable h = new Hashtable();
  foreach(string typename in typenames){
    h[typename] = 0;
  }
  foreach(object o in pets){
    if (o is Pet)
      h["Pet"] = ((int) h["Pet"]) + 1;
    if (o is Dog)
      h["Dog"] = ((int) h["Dog"]) + 1;
    if (o is Pug)
      h["Pug"] = ((int) h["Pug"]) + 1;
    if (o is Cat)
      h["Cat"] = ((int) h["Cat"]) + 1;
    if (o is Rodent)
      h["Rodent"] = ((int) h["Rodent"]) + 1;
    if (o is Gerbil)
      h["Gerbil"] = ((int) h["Gerbil"]) + 1;
    if (o is Hamster)
      h["Hamster"] = ((int) h["Hamster"]) + 1;
  }
  foreach(object o in pets)
```

```
      Console.WriteLine(o.GetType());
    foreach(string s in typenames)
      Console.WriteLine("{0} quantity: {1}",
        s, h[s]);
  }
} ///:~
```

There's a rather narrow restriction on **is**: You can compare it to a named type only, and not to a **Type** object. In the example above you might feel that it's tedious to write out all of those **is** expressions, and you're right. But there is no way to cleverly automate **is** by creating an **ArrayList** of **Type** objects and comparing it to those instead (stay tuned—you'll see an alternative). This isn't as great a restriction as you might think, because you'll eventually understand that your design is probably flawed if you end up writing a lot of **is** expressions.

Of course this example is contrived—you'd probably put a **static** data member in each type and increment it in the constructor to keep track of the counts. You would do something like that *if* you had control of the source code for the class and could change it. Since this is not always the case, RTTI can come in handy.

Using type retieval

It's interesting to see how the **PetCount.cs** example can be rewritten using type retrieval. The result is significantly cleaner:

```
//:c13:PetCount2.cs
//Compile with:
//csc Pets.cs PetCount2.cs
// Using type retrieval
using System;
using System.Collections;

public class PetCount2 {
  public static void Main(String[] args){
    ArrayList pets = new ArrayList();
    Type[] petTypes = {
      // Class literals:
      typeof(Pet),
      typeof(Dog),
      typeof(Pug),
      typeof(Cat),
      typeof(Rodent),
      typeof(Gerbil),
```

```
      typeof(Hamster)
    };

    Random r = new Random();
    for (int i = 0; i < 15; i++) {
      //Offset by 1 to eliminate Pet class
      Type t = petTypes[
        1 + r.Next(petTypes.Length - 1)];
      object o = Activator.CreateInstance(t);
      pets.Add(o);
    }

    Hashtable h = new Hashtable();
    foreach(Type t in petTypes){
      h[t] = 0;
    }
    foreach(object o in pets){
      Type t = o.GetType();
      foreach(Type mightBeType in petTypes){
        if (t == mightBeType ||
          t.IsSubclassOf(mightBeType)) {
          h[mightBeType] =
            ((int) h[mightBeType]) + 1;
        }
      }
    }
    foreach(object o in pets)
      Console.WriteLine(o.GetType());
    foreach(Type t in petTypes)
      Console.WriteLine("{0} quantity: {1}",
        t, h[t]);
  }
} ///:~
```

Here, the **typenames** array has been removed in favor of using the types directly as the **Hashtable** keys.

When the **Pet** objects are dynamically created, you can see that the random number is restricted so it is between one and **petTypes.length** and does not include zero. That's because zero refers to **Pet.class**, and presumably a generic **Pet** object is not interesting.

The loop that counts the different types needs to increment the count of the base classes (and interfaces) of the particular pet. Given a **Type** you can work in either direction: You can determine whether **Type aType** is a subtype of **Type maybeAncestor** by calling:

```
maybeAncestor.IsAssignableFrom(aType);
```

or you can call:

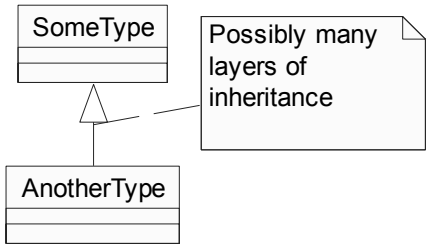
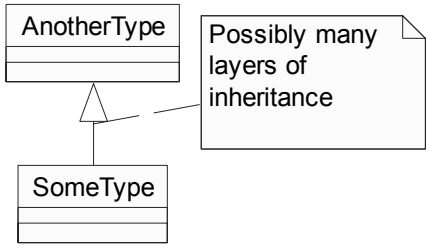
```
aType.IsSubclassOf(maybeAncestor);
```

In this example, we use the latter method to determine our count.

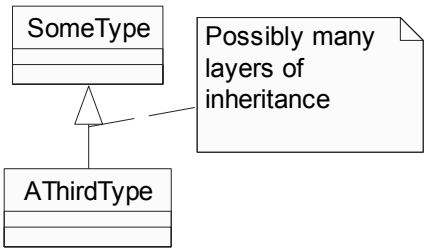
Given a **Type** and an object, you can use **Type.IsInstanceOfType( )**, passing in the object, as well. So, if:

```
Object someObject = new SomeType();
Object anotherObject = new AnotherType();
Object aThirdObject = new AThirdType();
Type someType = someObject.GetType();
Type anotherType = anotherObject.GetType();
```

Then:

Operation	If true:
`someType.isAssignableFrom (anotherType);`	
`someType.IsSubclassOf (anotherType);`	

```
someType.IsInstanceOfType
   (aThirdObject);
```



RTTI syntax

C# performs its RTTI using the **Type** object, even if you're doing something like a cast. The class **Type** also has a number of properties and methods that you can use to exploit RTTI.

First, you must get a reference to the appropriate **Type** object. One way to do this, as shown in the previous example, is to use a string and the **Type.GetType( )** method. This is convenient because you don't need an object of that type in order to get the **Type** reference. However, if you do already have an object of the type you're interested in, you can fetch the **Type** reference by calling a method that's part of the **object** root class: **GetType( )**. This returns the **Type** reference representing the actual type of the object. **Type** has some interesting methods, partially explored in the following example:

```
//:c13:ToyTest.cs
// Testing class Type.
using System;

interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}
class Toy {
  // Comment out or make less visible the following
  //default constructor to see MissingMethodException
  //thrown at (*1*)
  public Toy() {}
  public Toy(int i) {}
}

class FancyToy : Toy, HasBatteries,
Waterproof, ShootsThings {
  FancyToy():base(1) {}
```

```
}

public class ToyTest {
  public static void Main(){
    Type t = null;
    t = Type.GetType("FancyToy");
    PrintInfo(t);
    Type[] faces = t.GetInterfaces();
    foreach(Type iFace in faces){
      PrintInfo(iFace);
    }
    Type parent = t.BaseType;
    Object o = null;
    // Requires default constructor:
    Console.WriteLine("Creating " + parent);
    o = Activator.CreateInstance(parent); // (*1*)
    PrintInfo(o.GetType());
  }

  static void PrintInfo(Type t) {
    Console.WriteLine(
      "Class name: " + t.FullName
      + " is interface? ["
      + t.IsInterface + "]");
  }
} ///:~
```

You can see that **class FancyToy** is quite complicated, since it inherits from **Toy** and implements the **interface**s of **HasBatteries**, **Waterproof**, and **ShootsThings**. In **Main( )**, a **Type** reference is created and initialized to the **FancyToy Type** using **Type.GetType( )**.

The **Type.GetInterfaces( )** method returns an array of **Type** objects representing the interfaces that are contained in the **Type** object of interest.

If you have a **Type** object you can also ask it for its direct base class using the **BaseType** property. This, of course, returns a **Type** reference that you can further query. This means that, at run-time, you can discover an object's entire class hierarchy.

The **Activator.CreateInstance( )** can, at first, seem like just another way to clone an object. However, you can create a new object with **CreateInstance( )** *without* an existing object, as seen here, because there is no **Toy** object—only

parent, which is a reference to a **Type** object. This is a way to implement a "virtual constructor," which allows you to say "I don't know exactly what type you are, but create yourself properly anyway." In the example above, **parent** is just a **Type** reference with no further type information known at compile-time. And when you create a new instance, you get back an **object reference**. But that reference is pointing to a **Toy** object. Of course, before you can send any messages other than those accepted by **object**, you have to investigate it a bit and do some casting. In addition, in this scenario the class that's being created with **CreateInstance( )** must have a default constructor. In the next section, you'll see how to dynamically create objects of classes using any constructor, with the C# *reflection* API.

The final method in the listing is **PrintInfo( ),** which takes a **Type** reference and gets its name, including its namespace, from its **FullName** property and whether it's an interface with **IsInterface**.

The output from this program is:

```
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
creating Toy
Class name: Toy is interface? [false]
```

Thus, with the **Type** object you can find out just about everything you want to know about an object.

Reflection: run-time class information

If you don't know the precise type of an object, RTTI will tell you. However, there's a limitation: the type must be known at compile-time in order for you to be able to detect it using RTTI and do something useful with the information. Put another way, the compiler must know about all the classes you're working with for RTTI.

This doesn't seem like that much of a limitation at first, but suppose you're given a reference to an object that's not in your program space. In fact, the class of the object isn't even available to your program at compile-time. For example, suppose you get a bunch of bytes from another **AppDomain** or from a network connection and you're told that those bytes represent a class. Since the compiler

can't know about the class while it's compiling the code, how can you possibly use such a class?

In a traditional programming environment this seems like a far-fetched scenario. But as we move into a larger programming world there are important cases in which this happens. The first is component-based programming, in which you build projects using *Rapid Application Development* (RAD) in an application builder tool such as the Visual Designer in Visual Studio .NET. This is a visual approach to creating a program (which you see on the screen as a "form") by moving icons that represent components onto the form. These components are then configured by setting some of their properties at program time. This design-time configuration requires that any component be instantiable, that it exposes parts of itself, and that it allows its values to be read and set. In addition, components that handle GUI events must expose information about appropriate methods so that the RAD environment can assist the programmer in overriding these event-handling methods. Reflection provides the mechanism to detect the available methods and produce the method names.

And the Visual Designer and other "form builders" are just a step on the path towards visual programming. By using reflection to access a type's methods and the arguments to those methods, graphical editors can be used to specify significant amounts of program behavior.

Another compelling motivation for discovering class information at run-time is to provide the ability to create and execute objects on remote platforms across a network. This is called *Remoting* and it allows a C# program to have objects distributed across many machines. This distribution can happen for a number of reasons: For example, perhaps you're doing a computation-intensive task and you want to break it up and put pieces on machines that are idle in order to speed things up. In some situations you might want to place code that handles particular types of tasks (e.g., "Business Rules" in an *n*-tier architecture) on a particular machine, so that machine becomes a common repository describing those actions and it can be easily changed to affect everyone in the system. (This is an interesting development, since the machine exists solely to make software changes easy!) Along these lines, distributed computing also supports specialized hardware that might be good at a particular task—matrix inversions, for example—but inappropriate or too expensive for general purpose programming.

The class **Type** (described previously in this chapter) supports the concept of *reflection*, and there's an additional namespace, **System.Reflection,** with classes **EventInfo, FieldInfo**, **MethodInfo**, **PropertyInfo**, and **ConstructorInfo** (each of which inherit from **MemberInfo**). Objects of these

types are created at run-time to represent the corresponding member in the unknown class. You can then use the **ConstructorInfo**s to create new objects, read and modify fields and properties associated with **FieldInfo** and **PropertyInfo** objects, and the **Invoke( )** method to call a method associated with a **MethodInfo** object. In addition, you can call the convenience methods **Type.GetEvents( ), GetFields( )**, **GetMethods( )**, **GetProperties( )**, **GetConstructors( )**, etc., to return arrays of the objects representing the fields, methods, properties, and constructors. (You can find out more by looking up the class **Type** in your online documentation.) Thus, the class information for anonymous objects can be completely determined at run-time, and nothing need be known at compile-time.

It's important to realize that there's nothing magic about reflection. When you're using reflection to interact with an object of an unknown type, the CLR will simply look at the object and see that it belongs to a particular class (just like ordinary RTTI) but then, before it can do anything else, the **Type** object must be loaded. Thus, the assembly for that particular type must still be available to the CLR, either on the local machine or across the network (unless you are using the **System.Reflection.Emit** namespace to dynamically create types – a powerful capability that is beyond the scope of this book). So the true difference between RTTI and reflection is that with RTTI, the compiler opens and examines the assembly at compile-time. Put another way, you can call all the methods of an object in the "normal" way. With reflection, the assembly is unavailable at compile-time; it is opened and examined by the run-time environment.

Adding meta-information with attributes

How would you implement a mechanism whose behavior was applicable to a broad array of types? Examples of such "cross-cutting" concerns could include security, serialization, testing, and more esoteric things like the role of a class or method plays in implementing a design pattern.

You'd face two challenges: one would be associating your mechanism and its parameters with all the different types to which it applies, the second would be integrating, at the appropriate time, the custom mechanism into the behavior of the system. Attributes provide an efficient mechanism for the former, while reflection is used to help with the second challenge.

Attributes are just classes

An Attribute is just a class descended from class **Attribute**. This is a perfectly valid attribute:

```
//:c13:Meaningless1.cs
```

```
//Compile with csc /target:library Meaningless1.cs
using System;
public class Meaningless : Attribute {
}///:~
```

What makes **Attribute**s special is that every .NET language must support a special syntax so that **Attribute** types can be associated with programming elements such as classes, methods, parameters, assemblies, and so forth. In C#, this association is done by putting the name of the attribute in square brackets immediately preceding the declaration of the target. So, for instance:

```
[SerializableAttribute] class MyClass { }
```

associates the type **SerializableAttribute** with the class **MyClass**. Additionally, if there is no type with the exact name specified in the brackets and if there is a type with the name of form ***BracketName*Attribute**, that type will be associated. So:

[Serializable] class MyClass { } will first try to find an **Attribute** of type **Serializable**, but if such a type does not exist, type **SerializableAttribute** will be associated. This is a naming convention that is used throughout the .NET Framework SDK: The type is named **BracketNameAttribute**, and applied with the shorthand **[BracketName]**.

Associating an **Attribute** with a programming element with square brackets is said to be a *declarative* association.

Specifying an attribute's targets

Although Attributes can be associated with almost any programming element, a specific attribute generally only makes sense when applied to a subset: **SerializableAttribute**, which controls the ability of an object to be transformed into and from a **Stream**, only makes sense for classes, structs, enums, and delegates. **SecurityAttribute**, on the other hand, applies to assemblies, classes, structs, constructors, and methods.

The programming elements with which an Attribute can be associated are said to be the Attribute's *targets*. The compiler will produce an error if an Attribute is declaratively associated with an invalid target. How are target's specified? Why, with the **AttributeUsageAttribute**:

```
//:c13:Meaningless2.cs
//Compile with csc /target:library Meaningless2.cs
//A meaningless attribute
using System;
```

```
[AttributeUsage (AttributeTargets.Class)]
public class Meaningless : Attribute {
  public Meaningless(){
    Console.WriteLine("Meaningless created");
  }
}///:~
```

The **Meaningless** class declaration element is preceded by a declarative association of the **AttributeUsageAttribute**. The declaration specifies that **Meaningless** Attributes can be associated with class declaration elements. This allows us to write, for instance:

```
//:c13:Jellyfish.cs
//Compile with:
//csc /reference:Meaningless2.dll Jellyfish.cs
//A class with a Meaningless attribute
using System;

[Meaningless] class Jellyfish {
  //! [Meaningless] <-- generates "not valid"
  public static void Main(){
    new Jellyfish();
    Console.WriteLine("Jellyfish created");
  }
}///:~
```

Class **Jellyfish** is declared to have the **Meaningless** Attribute. In order for this class to compile, the compiler needs access to the **Meaningless** type, so this class must be compiled with a reference to the **Meaningless2.dll** generated in the previous sample.

Attribute arguments

Going back to the **AttributeUsageAttribute**, you'll see that it takes an argument, one or more values from the **AttributeTargets** enumeration. The **AttributeTargets** enum is defined as:

```
[Flags] public enum AttributeTargets{
Assembly = 1, Module = 2, Class = 4, Struct = 8,
Enum = 16, Constructor = 32, Method = 64,
```

```
Property = 128, Field = 256, Event = 512, Interface = 1024,
Parameter = 2048, Delegate = 4096, ReturnValue =- 8192, All
= 16383 };[2]
```

The **FlagsAttribute** makes **AttributeTargets** bitwise-combinable:

```
[AttributeUsage (AttributeTargets.Class |
AttributeTargets.Struct) ]
```

Or you can use **AttributeTargets.All** as shorthand for all the values.

It is the support for arguments that really make attributes shine. Not only can you associate the custom **Attribute** type with a target, you can associate custom design-time *data* with the target. The data must be a constant, typeof expression, or an array creation expression, but within those limits there is still a brand new area of interesting potential.

You can set an Attribute's arguments by overloading its constructor and by exposing public properties. Our **Meaningless3** Attribute uses both techniques:

```
//:c13:Meaningless3.cs
//Compile with csc /target:library Meaningless3.cs
//Demonstrates arguments to Attribute
using System;

[AttributeUsage (AttributeTargets.Class)]
public class Meaningless : Attribute {
  Type t;
  public Type TypeProperty{
    get { return t;}
    set { t = value;}
  }
  public Meaningless(string s){
    Console.WriteLine("Meaningless created");
  }
}///:~
```

To pass data in to an Attribute, you use a syntax which is quite different than anything we've seen:

```
[2]foreach(object o in
   Enum.GetValues(typeof(AttributeTargets))){
           Console.WriteLine("{0} : {1}", o, (int) o);
}
```

```
//:c13:Jellyfish2.cs
//Compile with:
//csc /reference:Meaningless3.dll Jellyfish2.cs
using System;
using System.Reflection;

[Meaningless("Some string",
    TypeProperty=typeof(Jellyfish))] class Jellyfish {
  public static void Main(){
    Console.WriteLine("In Main()");
    new Jellyfish();
    Console.WriteLine("Jellyfish created");
  }
}///:~
```

Things start off in a way that looks a little like a constructor, the attribute's typename, an open parenthesis, and then the constructor arguments (in this case, a single **string** variable). These constructor arguments have to be in order and a value must be provided for each expected argument in the constructor. Aside from the lack of the **new** keyword, this isn't far from what we're used to in C#. Then, *still within the open parenthesis*, comes a list of property name-value pairs. Not all properties have to be set in this list, the order in which properties are set is unimportant, the name of the property is not associated with any reference to an object and the list occurs within the same method-call-like parentheses pair that contains the constructor arguments.

Things get even stranger when we start investigating the behavior of Attributes. You may have noticed that our **Meaningless** Attributes have all written to the console in their constructor call, but if you've compiled and run the **Jellyfish** programs, you've seen that the **Meaningless** constructor has not written anything to the screen.

That's because Attributes aren't instantiated until they're retrieved! The Attribute arguments are actually "pickled" in the target assembly's metadata section and are only brought out of storage and used to instantiate the Attribute when and if special steps are taken to read the Attribute.

This is done to avoid versioning issues with Attributes and so that attribute objects are not created at compile time, which could have security implications. An **Attribute** type, like any other, has a public interface and a private implementation. The public interface of a type is only changed with great deliberation, but the private implementation should be absolutely transient—you should be able to change it on a whim. This raises the great problem of object

storage: The bytes that make up an object on Monday might not be able to recreate that object on Wednesday if on Tuesday a new assembly was installed with a different implementation.

One strategy for handling this is to figure that implementations don't change that often. This is the strategy Java uses: it assumes that bytes stored on Monday will be valid on Wednesday and, if on Wednesday this turns out to be an invalid assumption, Java throws an Exception while recreating the object. C#'s language design, though, is *clearly* influenced by Microsoft's experience with "DLL Hell," where the uncommon event of changing an implementation file, multiplied by tens of millions of users, became an expensive support issue. Thus, .NET's strategy for storing Attribute arguments: instead of instantiating the **Attribute** and storing its bytes, the arguments themselves are stored when the target is compiled on Monday. On Wednesday, as long as the **Attribute**'s assembly contains an appropriate constructor and properties, the **Attribute** is instantiated for the first time, using the arguments stored in the target assembly[3]. This section is very difficult to understand. Maybe if you expand it a bit, it will make more sense. Giving a concrete example would probably be a good idea.

The Global Assembly Cache

Attributes are also used to control the process by which shared assemblies are uniquely identified and made available to all programs that use them. This was covered very briefly in the discussion of namespaces in chapter 6, but the discussion of the process by which an assembly was uniquely named, signed, and installed had to be deferred until this discussion of attributes.

There are three major problems that the cryptographic approach to a global assembly cache solves:

1. How can I judge whether to install a component?
2. How can I ensure that installing one version of a component will not overwrite a version that is still needed by another program?
3. How can I know that an installed component hasn't been replaced by a malicious spoof?

---

[3] If, on the other hand, the **Attribute**'s assembly has had its public interface changed, .NET Framework v. 1.0.3705 fatally crashes the runtime, no matter what you do with try...catch and finally blocks. Hopefully, Microsoft will demonstrate the ease with which implementations can be updated and this won't be an issue by the time you read this!

All of these solutions are premised on the use of tools to manipulate the GAC, so we're not going to detail the structure of the directory \Windows\Assembly\GAC.

The *strong identity* of your assembly is determined by four things: the **[AssemblyProduct], [AssemblyVersion]** and **[AssemblyCulture]** attributes and the assembly's filename. The **AssemblyProductAttribute** takes a **string** that specifies the name of the assembly (which can be more elaborate than just a filename). The **AssemblyVersionAttribute** takes a **string** that should be a dotted number; the convention is that this is, from left-to-right, the major version, the minor version, the number of the official daily build, and an extra number for intraday emergency releases. The **AssemblyCultureAttribute** can specify culture-specific assemblies (discussed further in Chapter 14), when passed a blank string in its constructor, it creates an assembly that is "neutral" about culture.

In order to sign your assembly cryptographically, you must additionally specify the **AssemblyKeyFileAttribute**, which takes a **string** specifying the name of a file that contains both a public and private cryptographic key:

```
[assembly:AssemblyKeyFile("MyKeyFile.keys")]
```

This line specifies that the target of the attribute is not the element that immediately follows, but the assembly in which the attribute declaration is made.

The contents of **MyKeyFile.keys**, meanwhile, is created by running the Strong Naming Tool that ships with the .NET Framework SDK:

```
sn –k MyKeyFile.keys
```

The **sn** tool has a wide variety of case-sensitive command-line options.

The complete set of attributes for a strongly named class would look like this:

```
//:c13:StronglyNamedAttribute.cs
/* Compile with
csc /target:library StronglyNamedAttribute.cs
*/
using System;
using System.Reflection;

//Common name of product
[assembly:AssemblyProduct("Strongly Named Example")]
//First major release, built on day 180 of the project
[assembly:AssemblyVersion("1.0.180.0")]
```

```
//Location of public-private keyfile
[assembly:AssemblyKeyFile("MyKeyFile.keys")]
//Neutral culture
[assembly:AssemblyCulture("")]

[AttributeUsage (AttributeTargets.Class)]
public class SName : Attribute {
  Type t;
  public Type TypeProperty{
    get { return t;}
    set { t = value;}
  }
  public SName(){
    Assembly thisAssembly =
    Assembly.GetAssembly(typeof(SName));
    bool GACLoaded =
    thisAssembly.GlobalAssemblyCache;
    AssemblyName thisName = thisAssembly.GetName();
    string fName = thisName.FullName;
    Version v = thisName.Version;

    Console.WriteLine("{0} v. {1} from GAC: {2}",
      fName, v, GACLoaded);
  }
}///:~
```

There are several other **Assembly*Xxx*Attribute** classes that can embed other information data (such as copyright information) into an assembly, but do not influence the strong name of the assembly.

The **SName( )** constructor uses **Assembly.GetAssembly( )** to retrieve its own **Assembly**. The **Assembly.GlobalAssemblyCache** property specifies whether the **Assembly** was loaded from the GAC. The **AssemblyName** contains all of the strong name attributes, including version information and the public key.

After compilation, this assembly can be added to the GAC by running:

```
GACUtil /i StronglyNamedAttribute.dll
```

Once in the GAC, an assembly is available for runtime sharing; however, it is *not* automatically available for use as a compile-time reference. In other words, once you install **StronglyNamedAttribute.dll** in the GAC, you will *not* be able to switch directories and compile any class with

```
csc /reference:StronglyNamedAttribute.dll SomeClass.cs
```

This is initially confusing behavior, but makes sense when you consider the difference between the compilation use-case and the runtime loading use-case. Obviously, the compiler writers *could* have made the **/reference** switch check in the GAC if the referenced DLL was not in the current directory. But if the GAC contains multiple copies of **StronglyNamedAttribute.DLL**, which should be used by **SomeClass.cs** for referencing? The latest version? Probably, but not always! It is a not uncommon problem for one programmer on a team to install something and suddenly be using a different version of a shared library, which leads to the classic "It works okay on *my* machine," dilemma.

.NET is more restrictive by default: shared libraries *should* be placed within the development directory tree (and, of course, copies should be stored in the source-code control system). Although the **/reference** switch just refers to a filename; assemblies are bound to the complete strong name (filename, version, and public-key token).

If you've installed **StronglyNamedAssembly.dll** to the GAC, uninstall it with:

```
GACUtil /u StronglyNamedAssembly
```

Note that the uninstall command does not use the **.DLL** suffix. With a copy of **StronglyNamedAssembly.dll** in the local directory, compile this example:

```
//:c13:Jellyfish3.cs
//A class that refers to a strongly-named attribute
/*
   With StronglyNamedAttribute.DLL local compile with:
   csc /r:StronglyNamedAttribute.dll Jellyfish3.cs
*/
using System;
using System.Reflection;

[SName] class Jellyfish {
  public static void Main(){
    Console.WriteLine("In Main()");
    new Jellyfish();
    Console.WriteLine("Jellyfish created");
    try {
      //Trigger attribute instantiation
      Attribute.GetCustomAttribute
      (typeof(Jellyfish), typeof(SName));
    } catch (Exception e) {
```

```
      Console.WriteLine(e);
    }
  }
}///:~
```

Run **Jellyfish.exe**. **Attribute.GetCustomAttribute( )** will trigger the instantiation of the **SName** attribute. If you've deleted **StronglyNamedAssembly** from the GAC, the value of **GACLoaded** in the **SName( )** constructor will be **false**. Reinstall **StronglyNamedAssembly.DLL** in the GAC:

```
GACUtil /i StronglyNamedAssembly.DLL
```

Now when you run **Jellyfish.exe**, **GACLoaded** will be **true** and you are free to delete the local copy of **StronglyNamedAssembly.DLL**. Now, **Jellyfish.exe** is tightly bound to the version of **StronglyNamedAssembly** that is in the GAC. I think it would be valuable to go into more depth about what's happening behind the scenes in this section. I think the deeper the explanation, the easier it will be to see what's going on at the highest level. I've reread the section several times, and it's still not entirely clear what's happening. When run, the output will be similar to this (your public key will be different, of course):

```
In Main()
Jellyfish created
StronglyNamedAttribute, Version=1.0.180.0, Culture=neutral,
PublicKeyToken=24ced6b495827404 v. 1.0.180.0 from GAC: True
```

You can browse the contents of the GAC using Windows Explorer:

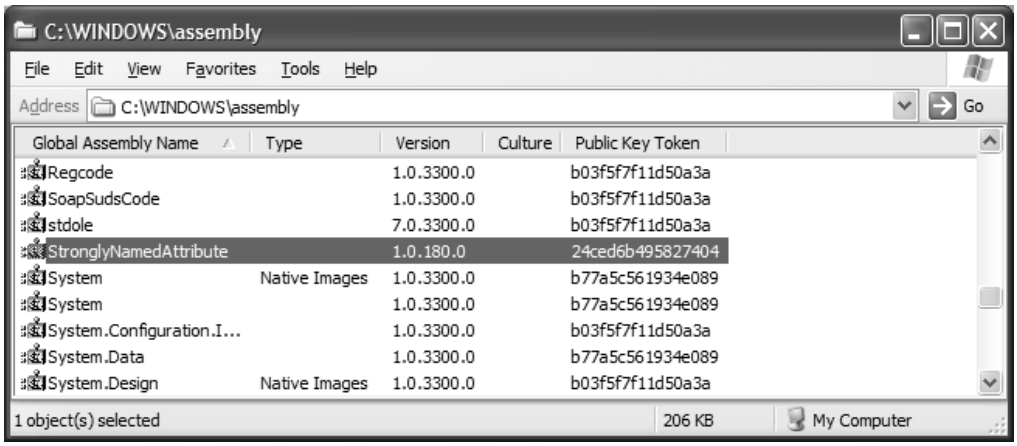


Figure 13-2: Windows Explorer allows you to view the GAC

If you use Visual Studio, you will see that it is possible to add references to .NET assemblies from a dialog. However, just adding an assembly to the GAC does not populate this dialog. If you are in the business of distributing libraries for other programmers, you'll probably want to make your DLLs available via this dialog rather than the generally-superior method of copying them into the project directory so that they're locally available.

Assemblies shown in this box are controlled by a path-based Registry key. So if prior to registering it in the GAC, you put **StronglyNamedAssembly.DLL** in, say, **c:\program files\mycompany\assemblies**, you would add to the Registry:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\Assemb
lyFolders\MyCompany]@="c:\program
files\mycompany\assemblies"
```

Figure 13-3 shows the results; a new registry value and the ability to reference the assembly from Visual Studio without additional XCOPY deployment of the library.

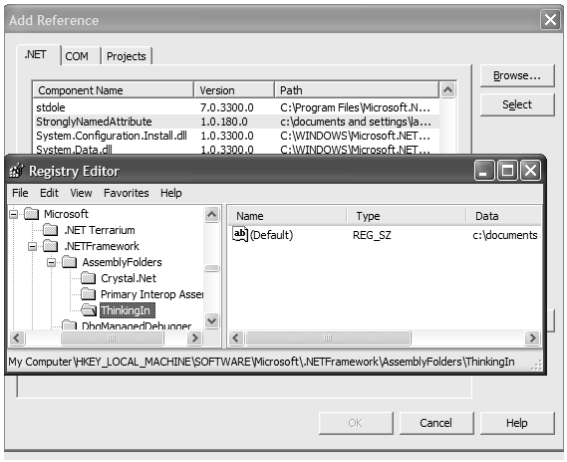


Figure 13-3: Adding an assembly to thte GAC

Designing with attributes

A common question among first-time attribute developers is "After I've written my **Attribute**, how do I automatically intercept it?" That is, attributes implemented in the .NET runtime such as **STAThreadAttribute** and **WebMethodAttribute** (discussed in Chapter 18) work to seamlessly modify their target's behavior. The belief is that there's some way to say "Get me all the targets of **Meaningless** currently in memory." There is not. The steps that make

a WebService as simple as adding **[WebMethod]** to a method only happen because Visual Studio checks for the attribute within the chain of code associated with compilation.[4]

Retrieving the Attributes associated with a target is done with the static methods **Attribute.GetCustomAttribute( )** and **Attribute.GetCustomAttributes( )**, which have overloads for each of the **AttributeTargets**. **Attribute.GetCustomAttribute( )** takes, as its final argument, the type of the **Attribute** in which you are interested.

So if attributes aren't magic – **Attribute** is just another class and there's no cloud of Attributes floating around in the execution space—what are the sorts of things you should do with them?

Attributes can target everything from assemblies to fields, which gives one clue. They are stored in their target's assembly's *metadata*, which gives another. Attributes are generally used, not to interfere with the target's state or behavior, but to work at a level outside those issues, perhaps modifying the context in which that function takes place or perhaps to control behavior at a time *other than* the target's runtime. Attributes are best used to tackle problems in *meta-programming* – programming whose subject is the task of programming or the execution of a program.

This example is a sketch of how Attributes might be used in a unit-testing framework. The idea is that testing could be made more rigorous if the code for a class or method was explicitly linked to its tests with a **[TestWith]** attribute. Then, a batch-oriented program could read an assembly to determine the appropriate tests, run them, and take the appropriate steps.

First, a simple **TestWithAttribute**:

```
//:c13:TestWithAttribute.cs
//Compile with
//csc /target:library TestWithAttribute.cs
//Associates a test class or method with its target
using System;

[AttributeUsage (AttributeTargets.Class |
```

[4] Actually, .NET's *context architecture* provides exactly the sort of dynamic interception described. This aspect of the CLR is a moving, undocumented target. The adventurous are referred to the **System.Runtime.Remoting.Contexts** namespace.

```
   AttributeTargets.Method)]
public class TestWithAttribute : Attribute {
  Type testType;

  public Type TestType{
    get { return testType;}
    set { testType = value;}
  }
  public TestWithAttribute(Type t){
    testType = t;
  }
}///:~
```

This attribute stores a **Type** that contains the testing code to exercise the **TestWithAttribute**'s target. In this next snippet, **[TestWith]** is applied to a trivial addition method:

```
//:c13:SomethingToTest.cs
//A class that could be tested
using System;

class SomethingToTest {
  [TestWith(typeof(AdditionTester))]
  public static int Add(int x, int y){
    return x + y;
  }
}///:~ (Example continues with AdditionTester.cs)
```

The **TestWithAttribute( )** constructor takes a reference to the **AdditionTester** type:

```
//:c13:AdditionTester.cs
//A dynamically applied test class
/* Compile with:
csc /reference:TestWithAttribute.dll /target:library
    SomethingToTest.cs AdditionTester.cs
*/
using System;

class AdditionTester {
  //Invoke all test methods
  public AdditionTester(){
    Console.WriteLine("Invoking test methods");
```

```
    if (TestPositiveNumbers()) {
      Console.WriteLine("Test passed");
    } else {
      Console.WriteLine("Test failed");
    }
  }
  bool TestPositiveNumbers(){
    int shouldBeFour = SomethingToTest.Add(2, 2);
    if (shouldBeFour != 4) {
      return false;
    } else {
      return true;
    }
  }
}///:~
```

When an instance of the **AdditionTester** stored in the **TestWithAttribute** is constructed, it exercises the target's methods; in this case, just confirming that 2 + 2 = 4. In a complete design for a testing framework, all testing classes would likely descend from a base class that used reflection and patterns such as *Template Method* to consistently apply test methods and react to the test methods' success or failure.

The final component of our **TestWithAttribute** example is **TestRunner**, a class which uses reflection and the **TestWithAttribute** but has no design-time knowledge of either the testing types or the targets of those tests:

```
//:c13:TestRunner.cs
//Demonstrates reflection & attributes
using System;
using System.Text.RegularExpressions;
using System.Reflection;

class TestRunner {
  //Usage: TestRunner TypeName,Assembly*
  public static void Main(string[] args){
    Regex pairPattern =
      new Regex(@"(?<tName>.*),(?<aName>.*)");
    foreach(string arg in args){
      Match m = pairPattern.Match(arg);
      string typeName = m.Groups["tName"].Value;
      string asmName = m.Groups["aName"].Value;
```

```
      new TestRunner(typeName, asmName);
    }
  }

  TestRunner(string typeName, string asmName){
    Assembly typeAsm = Assembly.Load(asmName);
    Type targetType = typeAsm.GetType(typeName);
    MethodInfo[] methods = targetType.GetMethods();

    foreach(MethodInfo method in methods){
      TestMethod(method);
    }
  }

  void TestMethod(MethodInfo methodInfo){
    TestWithAttribute twa =
      (TestWithAttribute)
      Attribute.GetCustomAttribute(
      methodInfo, typeof(TestWithAttribute));
    if (twa == null) {
      Console.WriteLine("No test defined for: " +
        methodInfo.Name);
    } else {
      Type testType = twa.TestType;
      Console.WriteLine("Testing {0} with {1}",
        methodInfo.Name, testType);
      Type[] noargs = new Type[0];
      ConstructorInfo defCon =
        testType.GetConstructor(noargs);
      defCon.Invoke(null);
    }
  }
}///:~
```

TestRunner's **Main( )** method is called with a command line argument of the form:

```
TestRunner TestTarget,TestTargetAssembly
```

The **Main( )** method loops over the command-line arguments, using a regular expression to parse the test target and assembly names. **Main( )** passes each type-and-assembly pair to a **TestRunner( )** constructor.

The **TestRunner( )** constructor shows how easy it is to use reflection: It dynamically loads the specified assembly, the to-be-tested type, and retrieve's all of that type's methods. For each method, it calls **TestRunner.TestMethod( )**.

TestMethod( ) shows how to retrieve Attributes with **Attribute.GetCustomAttribute( )**. The first argument is the passed in **MethodInfo**, the second argument the **Type** of the Attribute we are trying to retrieve, in this case **typeof(TestWithAttribute)**. If the method is not associated with the specified **Attribute**, the variable **twa** is null and a message is written to the screen.

If, on the other hand, there is a **TestWithAttribute** associated with the current method, the act of retrieving the **TestWithAttribute** will trigger the creation of the **TestWithAttribute**. So when the **TestRunner** gets to **SomethingToTest.Add( )**, a **TestWithAttribute** is created with a constructor argument of **typeof(AdditionTester)**. The constructor assigns the **TestWithAttribute.TestType** property to **typeof(AdditionTester)**.

After writing a message to the screen, **TestRunner.TestMethod( )** retrieves the **TestWithAttribute.TestType** and uses reflection to retrieve its no-args constructor. **Type.GetConstructor( )** takes a **Type][ ]** representing the argument list, since we want the default constructor we just pass in an empty array.

Finally, **TestMethod** uses **ConstructorInfo.Invoke( )** to instantiate an object of type **AdditionTester**. As discussed previously, the **AdditionTester( )** constructor takes care of the business of running the test methods.

The output of:

```
TestRunner SomethingToTest,SomethingToTest
```

is:

```
No test defined for: GetHashCode
No test defined for: Equals
No test defined for: ToString
Testing Add with AdditionTester
Invoking test methods
Test passed
No test defined for: GetType
```

Which illustrates the potential benefit of using Attributes for unit-testing: The important question "What is the test coverage for this type and is that coverage complete?" is answered by the **Type** itself via its **TestWithAttributes**.

Beyond objects with aspects

Attributes allow you to perform meta-programming on a broad range of targets, bypassing inheritance as the only means of organizing types across a program or system. However, you still have to explicitly associate an **Attribute** with each of its targets and there is not an automatic way to "inject" the **Attribute**'s behavior into the program's behavior. *Aspect-Oriented Programming* is an emerging technology that eliminates both of these restrictions. Aspect-Oriented Programming is generating a buzz similar to the buzz that surrounded Object-Oriented Programming in the late 80s. Whether it's going to emerge as an important programming paradigm is by no means certain, but it's certainly something that should be on your radar screen.

At the moment, there are no aspect-oriented languages implemented on .NET. This is likely to change soon, as .NET's value proposition of "Any language, one platform," makes it fertile ground for new approaches to programming. Additionally, the .NET Framework Library contains classes beyond the scope of this book for emitting code, dynamic compilation, and programmatic type extension; .NET is potentially the most significant contribution to the development of new computer languages in 25 years. That would be nice; the rise of object-oriented imperative languages has had important benefits (the patterns movement could not have happened without the widespread adoption of the vocabulary of objects) but there has been a loss of awareness of the diversity and potential of alternate paradigms. Aspect-orientation, declarative, functional, and data-flow programming all have things to offer us today, and these are just some obvious areas. Hopefully, the commercial use of .NET will not blind visionaries to the value of the .NET infrastructure for research and development of programming technology.

Summary

RTTI allows you to discover type information from an anonymous base-class reference. Thus, it's ripe for misuse by the novice since it might make sense before polymorphic method calls do. For many people coming from a procedural background, it's difficult not to organize their programs into sets of **switch** statements. They could accomplish this with RTTI and thus lose the important value of polymorphism in code development and maintenance. The intent of C# is that you use polymorphic method calls throughout your code, and you use RTTI only when you must.

However, using polymorphic method calls as they are intended requires that you have control of the base-class definition because at some point in the extension of your program you might discover that the base class doesn't include the method you need. If the base class comes from a library or is otherwise controlled by someone else, a solution to the problem is RTTI: You can inherit a new type and add your extra method. Elsewhere in the code you can detect your particular type and call that special method. This doesn't destroy the polymorphism and extensibility of the program because adding a new type will not require you to hunt for switch statements in your program. However, when you add new code in your main body that requires your new feature, you must use RTTI to detect your particular type.

Putting a feature in a base class might mean that, for the benefit of one particular class, all of the other classes derived from that base require some meaningless stub of a method. This makes the interface less clear and annoys those who must override abstract methods when they derive from that base class. For example, consider a class hierarchy representing musical instruments. Suppose you wanted to clear the spit valves of all the appropriate instruments in your orchestra. One option is to put a **ClearSpitValve( )** method in the base class **Instrument**, but this is confusing because it implies that **Percussion** and **Electronic** instruments also have spit valves. RTTI provides a much more reasonable solution in this case because you can place the method in the specific class (**Wind** in this case), where it's appropriate. However, a more appropriate solution is to put a **PrepareInstrument( )** method in the base class, but you might not see this when you're first solving the problem and could mistakenly assume that you must use RTTI.

RTTI will sometimes solve efficiency problems. If your code nicely uses polymorphism, but it turns out that one of your objects reacts to this general purpose code in a horribly inefficient way, you can pick out that type using RTTI and write case-specific code to improve the efficiency. Be wary, however, of programming for efficiency too soon. It's a seductive trap. It's best to get the program working *first*, then decide if it's running fast enough, and only then should you attack efficiency issues—with a profiler.

Finally, Attributes and RTTI are used to associate metadata with a variety of elements in .NET's programming structure. Attributes are best used to either modify the context of execution or for the development of systems concerned with the programming structure itself. Attributes such as **[STAThread]** and **[DLLImport]** are examples of context-changing attributes, which modify the runtime environment of their targets. The **TestWithAttribute** sketched in this

chapter is an example of the second type of attribute, which concerns itself with something other than the runtime behavior of the target.

Exercises

1. Add **Rhomboid** to **Shapes.cs**. Create a **Rhomboid**, upcast it to a **Shape**, then downcast it back to a **Rhomboid**. Try downcasting to a **Circle** and see what happens.

2. Modify the previous exercises so that it uses is to check the type before performing the downcast. Now try it using as to perform the downcast but without checking.

3. Modify **Shapes.cs** so that it can "highlight" (set a flag) in all shapes of a particular type. The **ToString( )** method for each derived **Shape** should indicate whether that **Shape** is "highlighted."

4. Modify **SweetShop.cs** so that each type of object creation is controlled by a command-line argument. That is, if your command line is "**SweetShop Candy**," then only the **Candy** object is created. Notice how you can control which class objects are loaded via the command-line argument.

5. Add a new type of **Pet** to **PetCount2.cs**. Verify that it is created and counted correctly in **Main( )**.

6. Write a program that takes a class name from the command-line and prints all the objects in that class's inheritance hierarchy.

7. Create a new container that uses a **private ArrayList** to hold the objects. Capture the type of the first object placed into your container. From then on, allow the user to only insert objects of that type.

8. Write a **LastModifiedAttribute** that can be applied to a method. The attribute should specify the date and programmer who last touched the method and possibly an enumerated value of why the method was changed (new feature, defect correction, etc.). Write a program that loads an assembly and lists the classes and methods, sorted by their last-modified date.

9. Extend the previous example so that it works on all the assemblies in a given directory and can be filtered so that it only prints those classes and methods modified in the previous day, week, month, etc.

10. Extend the previous example so that the data is output to a file in a format appropriate for analyzing the activity of a development project (which classes are most active, which methods have been changed the most, etc.).

14: Programming Windows Forms

A fundamental design guideline is "make simple things easy, and difficult things possible." A variation on this is called "the principle of least astonishment," which essentially says: "don't surprise the user."

There have been two antagonistic forces at work in the Windows GUI programming world – the ease of use epitomized by Visual Basic and the control available to C programmers using the Win32 API. Reconciling these forces is one of the great achievements of the .NET Framework. Visual Studio .NET provides a VB-like toolset for drawing forms, dragging-and-dropping widgets onto them, and rapidly specifying their properties, event, and responses. However, this easy-to-use tool sits on top of Windows Forms and GDI+, systems whose object-oriented design, consistency, and flexibility are unsurpassed for creating rich client interfaces.

The aim of this chapter is to give a good understanding of the underlying concepts at play in implementing graphical user interfaces and to couple those concepts with concrete details on the most commonly-used widgets.

For Java programmers, the ease with which rich, highly responsive interfaces and custom components can be created in C# will come as a revelation. Java's UI models have been based on the premise that operating system integration is superfluous to the task of creating program interfaces. This is absurd, if for no other reason than its violation of the guidelines mentioned at the beginning of this chapter. Sun's bid to commoditize the operating system has utterly failed. While it may be that not *every* application requires OS-specific integration, it's equally obvious that *some* applications do. In addition to Windows, there remain

solid markets for applications that take advantage of the unique capabilities of the Macintosh, Palm, Linux, and, yes, Solaris operating systems.

The power of Windows programming requires a thorough understanding of another C# language feature, the delegate.

Delegates

One of C#'s most interesting language features is its support for *delegates*, an object-oriented, method type. The line:

```
delegate string Foo(int param);
```

is a *type declaration* just as is:

```
class Bar{ }
```

And just as to be useful a class type has to be instantiated (unless it just has **static** data and methods), so too must delegate types be instantiated to be of any use. A delegate can be instantiated with any method that matches the delegate's type signature. Once instantiated, the delegate reference can be used directly as a method. Delegates are object-oriented in that they can be bound not just to static methods, but to instance methods; in doing so, a delegate will execute the specified method on the designated object. A simple example will show the basic features of delegates:

```
//:c14:Profession.cs
//Declaration and instantiation of delegates
using System;

delegate void Profession();

class ProfessionSpeaker {

  static void StaticSpeaker(){
    Console.WriteLine("Medicine");
  }

  static int doctorIdCounter = 0;
  int doctorId = doctorIdCounter++;

  void InstanceSpeaker(){
    Console.WriteLine("Doctor " + doctorId);
  }
```

```
  int DifferentSignature(){
    Console.WriteLine("Firefighter");
    return 0;
  }

  public static void Main(){
    //declare delegate reference (== null)
    Profession p;
    //instantiate delegate reference
    p = new Profession(
      ProfessionSpeaker.StaticSpeaker);
    p();

    ProfessionSpeaker s1 = new ProfessionSpeaker();
    ProfessionSpeaker s2 = new ProfessionSpeaker();

    //"instantiate" to specific instances
    Profession p1 = new Profession(
      s1.InstanceSpeaker);
    Profession p2 = new Profession(
      s2.InstanceSpeaker);

    p1();
    p2();

    //Won't compile, different signature
    //Profession p3 = new Profession(
    //   s2.DifferentSignature);
  }
}///:~
```

The **Profession** delegate type is declared to take no parameters and to return **void**. The **ProfessionSpeaker** has two methods with this signature: a **static StaticSpeaker( )** method and an instance method called **InstanceSpeaker( )**. **ProfessionSpeaker** has a static **doctorIdCounter** which is incremented every time a new **ProfessionSpeaker** is instantiated; thus **InstanceSpeaker( )** has different output for each **ProfessionSpeaker** instance.

ProfessionSpeaker.Main( ) declares a delegate reference **Profession p**. Like all declared but not initialized variables, **p** is null at this point. The next line is the delegate's "constructor" and is of the form:

```
new DelegateTypeName( NameOfMethodToDelegate );
```

This first delegate is instantiated with the **StaticSpeaker( )** method (note that the value passed to the delegate "constructor" is just the *name* of the method without the parentheses that would specify an actual method *call*).

Once the delegate is initialized, the reference *variable* acts as if it was an in-scope method *name*. So the line **p( )** results in a call to **ProfessionSpeaker.StaticSpeaker( )**.

Main( ) then instantiates two new **ProfessionSpeaker**s; one of which will have a **doctorId** of 0 and the other of 1. We declare two new **Profession** delegates, but this time use a slight different form:

```
new DelegateTypeName( objectReference.MethodName );
```

You can think of this as passing "the instance of the method" associated with the **objectReference**, even though "that's not how it really is" in memory (there's a copy of each piece of *data* for an instance, but *methods* are not duplicated). The two delegates **p1** and **p2** can now be used as proxies for the methods **s1.InstanceSpeaker( )** and **s2.InstanceSpeaker( )**.

The delegated-to method must have the exact parameters *and return type* of the delegate type declaration. Thus, we can't use **DifferentSignature( )** (which returns an **int**) to instantiate a **Profession** delegate.

When run, the calls to the delegates **p, p1**, and **p2** result in this output:

```
Medicine
Doctor 0
Doctor 1
```

Designing With Delegates

Delegates are used extensively in Windows Forms, .NET's object-oriented framework for GUI programming. However, they can be effectively used in any situation where variables in behavior are more important than variables in state. We've already talked about the Strategy pattern (#ref#):

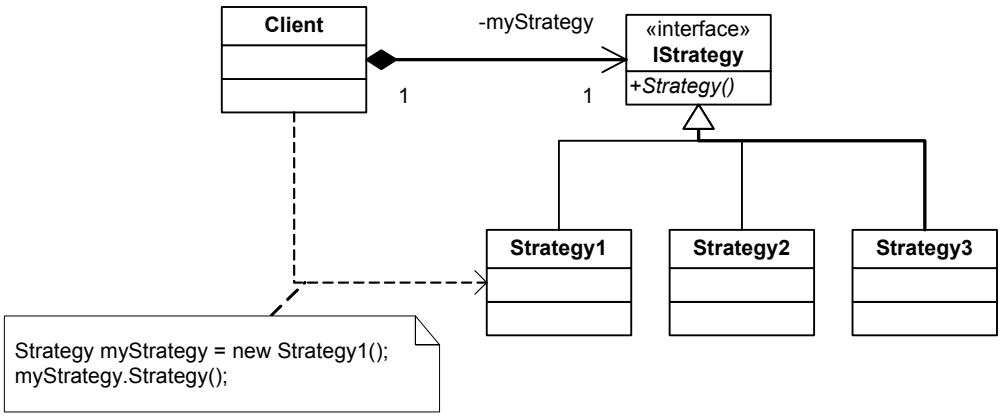


Figure 14-1: Interfaces are one way to implement the Strategy pattern

Delegates provide an alternate design:

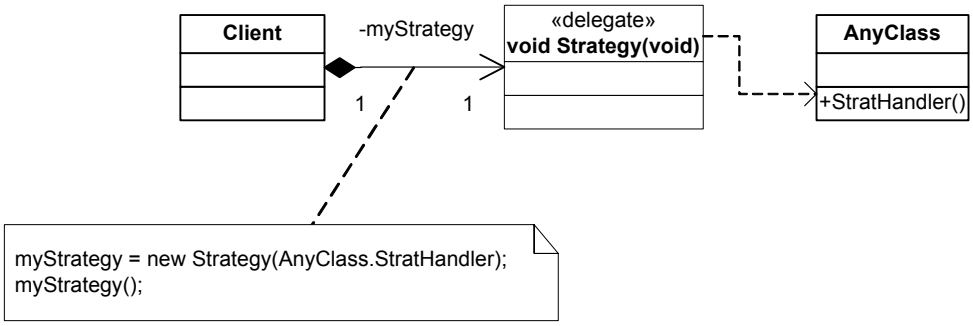


Figure 14-2: Delegates provide a slightly easier alternative

The Strategy pattern explicitly binds the implementation to its use satisfying the contract of the interface **IStrategy**; any class which will be used as a **Strategy** must declare itself as **: IStrategy** and implement the method **void Strategy( )**. With delegates, there is no explicit binding of the handler to the delegate type; a new programmer coming to the situation wouldn't necessarily know that **AnyClass.StratHandler( )** was being used to instantiate a **Strategy** delegate. This might lead to trouble: the new programmer could change the behavior of **StratHandler( )** in a way inappropriate to its use as a **Strategy** delegate and the change would not be caught by either the compiler or by **AnyClass**'s unit tests; the problem wouldn't appear until **Client**'s unit tests were run.

On the other hand, the delegate model is significantly less typing and can delegate handling to a method in a type for which you don't have source code (although there's no obvious scenario that would call for that).

Multicast delegates

Delegates that return type void may designate a *series* of methods that will be invoked when the delegate is called. Such delegates are called *multicast delegates*. Methods can be added and remove from the call chain by using the overloaded += and -= operators. Here's an example:

```
//:c14:Multicast.cs
//Demonstrates multicast delegates
using System;

class Rooster {
  internal void Crow(){
    Console.WriteLine("Cock-a-doodle-doo");
  }
}

class PaperBoy {
  internal void DeliverPapers(){
    Console.WriteLine("Throw paper on roof");
  }
}

class Sun {
  internal void Rise(){
    Console.WriteLine("Spread rosy fingers");
  }
}

class DawnRoutine {
  delegate void DawnBehavior();
  DawnBehavior multicast;

  DawnRoutine(){
    multicast = new DawnBehavior(new Rooster().Crow);
    multicast += new DawnBehavior(
      new PaperBoy().DeliverPapers);
    multicast += new DawnBehavior(new Sun ().Rise);
  }

  void Break(){
```

```
    multicast();
  }

  public static void Main(){
    DawnRoutine dr = new DawnRoutine();
    dr.Break();
  }
}///:~
```

After declaring three classes (**Rooster**, **PaperBoy**, and **Sun**) that have methods associated with the dawn, we declare a **DawnRoutine** class and, within its scope, declare a **DawnBehavior** delegate type and an instance variable **multicast** to hold the instance of the delegate. The **DawnRoutine( )** constructor's first line instantiates the delegate to the **Crow( )** method of a **Rooster** object (the garbage collector is smart enough to know that although we're using an *anonymous* instance of **Rooster**, the instance will not be collected as long as the delegate continues to hold a reference to it).

New instances of **DawnBehavior** are instantiated with references to "instances" of **PaperBoy.DeliverPapers( )** and **Sun.Rise( )**. These DawnBehavior delegates are added to the **multicast** delegate with the += operator. **Break( )** invokes **multicast( )** once, but that single call in turn is *multicast* to all the delegates:

```
Cock-a-doodle-doo
Throw paper on roof
Spread rosy fingers
```

Multicast delegates are used throughout Windows Forms to create event-handling chains, each of which is responsible for a particular aspect of the total desired behavior (perhaps display-oriented, perhaps logical, perhaps something that writes to a logfile).

Multicast delegates are similar to the *Chain of Responsibility* design pattern:

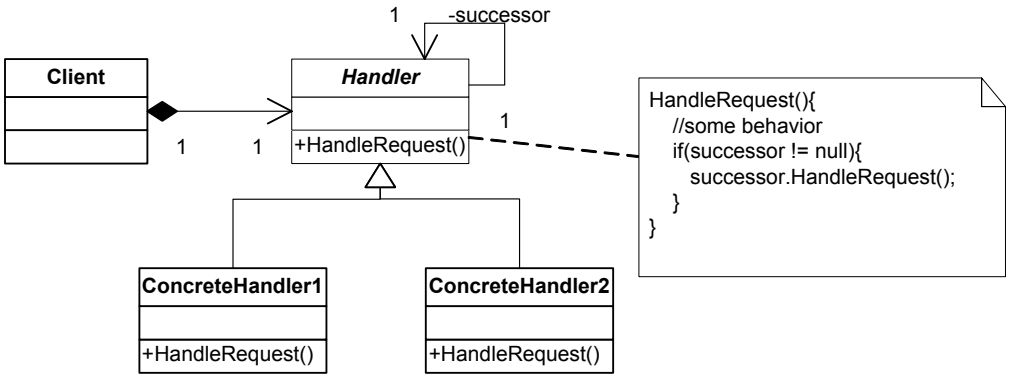


Figure 14-3: The Chain of Responsibility design pattern

The one difference is that the *Chain of Responsibility* is usually implemented so that an individual handler can decide to end the processing chain (that is, a handler may decide not to call its **successor**'s **HandleRequest( )** method). It is not possible for a delegated-to method to have this sometimes-desired option, so there may be times when you implement a *Chain of Responsibility* within an interface on which you create single or multicast delegates:

```
//:c14:DelegatedChainOfResponsibility.cs
//Shows polymorphic delegation, plus CoR pattern
using System;

interface Handler{
  void HandleRequest();
}

class ConcreteHandler1 : Handler {
  Random r = new Random();
  Handler successor = new ConcreteHandler2();

  public void HandleRequest() {
    if (r.NextDouble() > 0.5) {
      Console.WriteLine("CH1: Handling incomplete");
      successor.HandleRequest();
    } else {
      Console.WriteLine("CH1: Handling complete");
    }
  }
}
```

```
class ConcreteHandler2 : Handler {
  public void HandleRequest() {
    Console.WriteLine("H2 handling");
  }
}

class Client {
  delegate void PolyDel();

  private static void AlsoHandler(){
    Console.WriteLine("AlsoHandler");
  }

  public static void Main(){
    //Note upcast
    Handler h = new ConcreteHandler1();
    PolyDel del = new PolyDel(h.HandleRequest);
    del += new PolyDel(AlsoHandler);

    del();
  }
}///:~
```

The **Handler** interface and its two subtypes implement the *Chain of Responsibility* pattern. **ConcreteHandler1** "flips a coin" in the form of a random number and either calls **successor.HandleRequest( )** or not. The **PolyDel** delegate declared within **Client** matches the interface in **Handler**, so when a new **Handler** is instantiated in **Client.Main( )**, there is no problem in using that handle's **HandleRequest** method as the argument to a new **PolyDel** delegate. Since delegates are object-oriented, there is no problem using a handle that's been upcast to an abstract data type in the construction of a delegate. When **del( )** is invoked, half the time the **ConcreteHandler1** will forward a call to its **ConcreteHandler2 successor** and half the time not. Because **PolyDel** is a multicast handler, though, **AlsoHandler( )** will always be called.

Events

The behavior of a multicast delegate can be interpreted as a single event (the multicast method call) causing multiple behaviors (the calls on the various delegated-to methods). However, with a normal delegate, the delegated-to method has no way of knowing anything about the context in which it was

invoked. In our **Multicast** example, the **Sun.Rise( )** method (say) could not have affected any change in the **DawnRoutine** object that invoked it. By convention, multicast delegates that require context should be of type:

```
delegate void DelegateName(
      object source, EventArgsSubtype x);
```

The **EventArgs** class is defined in the **System** namespace and, by default, contains nothing but a static readonly property **Empty** that is defined as returning an **EventArg** equivalent to an **EventArg** created with a no-args constructor (in other words, a generic, undistinguishable, and therefore "Empty" argument).

Subtypes of **EventArgs** are expected to define and expose properties that are the most likely important pieces of context. For instance, the **DawnBehavior** delegate might be paired with a **DawnEventArgs** object that contained the weather and time of sunrise.

If a class wishes to define a multicast delegate of this sort and expose it publicly as a property, the normal C# syntax would be:

```
void DawnDelegate(object source, DawnEventArgs dea);
class DawnEventArgs : EventArgs{}

class DawnBehavior{
   private DawnDelegate d;
   public DawnDelegate DawnEvent{
      get { return d; }
      set { d = value; }
   }
}
```

A shortcut is to simply declare an *event property*:

```
public event DawnDelegate DawnEvent;
```

You still have to define the delegate and the subtype of **DawnEventArgs** and there is no difference in behavior between a public multicast delegate exposed as a normal property and one exposed as an event property. However, event properties may be treated differently by developer tools such as the doc-comment generator or visual builder tools such as the one in Visual Studio.NET.

This example shows event handlers that modify their behavior depending on the context:

```
//:c14:EventProperty.cs
using System;

delegate void DawnDelegate(
  object source, DawnEventArgs dea);

enum Weather {
  sunny, rainy
};

class DawnEventArgs : EventArgs {
  //Hide base class constructor
  private DawnEventArgs() : base(){}
  public DawnEventArgs(Weather w){
    this.w = w;
  }

  private Weather w;
  public Weather MorningWeather{
    set { w = value;}
    get { return w;}
  }
}

class DawnBehavior {
  public event DawnDelegate DawnEvent;

  public static void Main(){
    DawnBehavior db = new DawnBehavior();
    db.DawnEvent = new DawnDelegate(
      new Rooster().Crow);
    db.DawnEvent += new DawnDelegate(
      new Sun ().Rise);
    DawnEventArgs dea =
      new DawnEventArgs(Weather.sunny);
    db.DawnEvent(typeof(DawnBehavior), dea);
    dea = new DawnEventArgs(Weather.rainy);
    db.DawnEvent(typeof(DawnBehavior), dea);
  }
}
```

```
class Rooster {
  internal void Crow(object src, DawnEventArgs dea){
    if (dea.MorningWeather == Weather.sunny) {
      Console.WriteLine("Cock-a-doodle-doo");
    } else {
      Console.WriteLine("Sleep in");
    }
  }
}

class Sun {
  internal void Rise(object src, DawnEventArgs dea){
    if (dea.MorningWeather == Weather.sunny) {
      Console.WriteLine("Spread rosy fingers");
    } else {
      Console.WriteLine("Cast a grey pall");
    }
  }
}///:~
```

In this example, the **DawnEvent** is created in the static **Main( )** method, so we couldn't send **this** as the source nor does passing in the **db** instance seem appropriate. We could pass **null** as the source, but passing null is generally a bad idea. Since the event is created by a static method and a static method is associated with the class, it seems reasonable to say that the source is the type information of the class, which is retrieved by **typeof(DawnBehavior)**.

Recursive traps

Conceptually, event-driven programs are asynchronous – when an event is "fired" (or "raised" or "sent"), control returns to the firing method and, sometime in the future, the event handler gets called. In reality, C#'s events are synchronous, meaning that control does not return to the firing method until the event handler has completed. This conceptual gap can lead to serious problems. If the event handler of event *X* itself raises events, and the handling of these events results in a new event *X*, the system will recurse, eventually either causing a stack overflow exception or exhausting some non-memory resource.

In this example, a utility company sends bills out, a homeowner pays them, which triggers a new bill. From a conceptual standpoint, this should be fine, because the payment and the new bill are separate events.

```
//:c14:RecursiveEvents.cs
```

```
//Demonstrates danger in C# event model
using System;

delegate void PaymentEvent(object src, BillArgs ea);

class BillArgs {
  internal BillArgs(double c){
    cost = c;
  }
  public double cost;
}

abstract class Bookkeeper {
  public event PaymentEvent Inbox;

  public static void Main(){
    Bookkeeper ho = new Homeowner();
    UtilityCo uc = new UtilityCo();

    uc.BeginBilling(ho);
  }

  internal void Post(object src, double c){
    Inbox(src, new BillArgs(c));
  }
}

class UtilityCo : Bookkeeper {
  internal UtilityCo(){
    Inbox += new PaymentEvent(this.ReceivePmt);
  }

  internal void BeginBilling(Bookkeeper bk){
    bk.Post(this, 4.0);
  }

  public void ReceivePmt(object src, BillArgs ea){
    Bookkeeper sender = src as Bookkeeper;
    Console.WriteLine("Received pmt from " +
      sender);
    sender.Post(this, 10.0);
```

```
  }
}

class Homeowner : Bookkeeper {
  internal Homeowner(){
    Inbox += new PaymentEvent(ReceiveBill);
  }
  public void ReceiveBill(object src, BillArgs ea){
    Bookkeeper sender = src as Bookkeeper;
    Console.WriteLine("Writing check to " +
      sender + " for " + ea.cost);
    sender.Post(this, ea.cost);
  }
}///:~
```

First, we declare a delegate type called **PaymentEvent** which takes as an argument a **BillArgs** reference containing the amount of the bill or payment. We then create an abstract **Bookkeeper** class with a **PaymentEvent** event called **Inbox**. The **Main( )** for the sample creates a **HomeOwner**, a **UtilityCo,** and passes the reference to the **HomeOwner** to the **UtilityCo** to begin billing. **Bookkeeper** then defines a method called **Post( )** which triggers the **PaymentEvent( );** we'll explain the rationale for this method in a little bit.

UtilityCo.BeginBilling( ) takes a **Bookkeeper** (the homeowner) as an argument. It calls that **Bookkeeper**'s **Post( )** method, which in turn will call that **Bookkeeper**'s **Inbox** delegate. In the case of the **Homeowner**, that will activate the **ReceiveBill( )** method. The homeowner "writes a check" and **Post( )**s it to the source. If events were asynchronous, this would not be a problem.

However, when run, this will run as expected for several hundred iterations, but then will crash with a stack overflow exception. Neither of the event handlers (**ReceiveBill( )** and **ReceivePayment( )**) ever returns, they just recursively call each other in what would be an infinite loop but for the finite stack. More subtle recursive loops are a challenge when writing event-driven code in C#.

Perhaps in order to discourage just these types of problems, event properties differ from delegates in one very important way: An event can only be invoked by the very class in which it is declared; even descendant types cannot directly invoke an event. This is why we needed to write the **Post( )** method in **Bookkeeper**, **HomeOwner** and **UtilityCo** cannot execute **Inbox( )**, attempting to do so results in a compilation error.

This language restriction is a syntactical way of saying "raising an event is a big deal and must be done with care." Event-driven designs may require multiple threads in order to avoid recursive loops (more on this in Chapter 16). Or they may not. This restriction on events does not force you into any particular design decisions – as we showed in this example, one can simply create a public proxy method to invoke the event.

The genesis of Windows Forms

While C# events are not asynchronous, "real" Windows events are. Behind the scenes, Windows programs have an event loop that receives unsigned integers corresponding to such things as mouse movements and clicks and keypresses, and then say "If that number is *x*, call function *y*." This was state-of-the-art stuff in the mid-1980s before object-orientation became popular. In the early 1990s, products such as Actor, SQL Windows, Visual Basic, and MFC began hiding this complexity behind a variety of different models, often trading off object-oriented "purity" for ease of development or performance of the resulting application.

Although programming libraries from companies other than Microsoft were sometimes superior, Microsoft's libraries always had the edge in showcasing Windows latest capabilities. Microsoft parlayed that into increasing market share in the development tools category, at least until the explosion of the World Wide Web in the mid-1990s, when the then-current wisdom about user interfaces (summary: "UIs must consistently follow platform standards, and UIs must be fast") was left by the wayside for the new imperative ("All applications must run inside browsers").

One of the programming tools that had difficulty gaining marketshare against Microsoft was Borland's Delphi, which combined a syntax derived from Turbo Pascal, a graphical builder a la Visual Basic, and an object-oriented framework for building UIs. Delphi was the brainchild of Anders Hejlsberg, who subsequently left Borland for Microsoft, where he developed the predecessor of .NET's Windows Forms library for Visual J++. Hejlsberg was, with Scott Wiltamuth, one of the chief designers of the C# language and C#'s delegates trace their ancestry to Delphi. (Incidentally, Delphi remains a great product and is now, ironically, the best tool for programming native Linux applications!)

So Windows Forms is an object-oriented wrapper of the underlying Windows application. The doubling and redoubling of processor speed throughout the 1990s has made any performance hit associated with this type of abstraction irrelevant; Windows Forms applications translate the raw Windows events into

calls to multicast delegates (i.e., **event**s) so efficiently that most programmers will never have a need to side-step the library.

Creating a Form

With Windows Forms, the static Main() method calls the static method Application.Run(), passing to it a reference to a subtype of **Form**. All the behavior associated with creating, displaying, closing, and otherwise manipulating a Window (including repainting, a finicky point of the "raw" Windows API) is in the base type **Form** and need not be of concern to the programmer. Here's a fully functional Windows Form program:

```
//:c14:FirstForm.cs
using System.Windows.Forms;

class FirstForm : Form {
  public static void Main(){
    Application.Run(new FirstForm());
  }
}///:~
```

that when run produces this window:

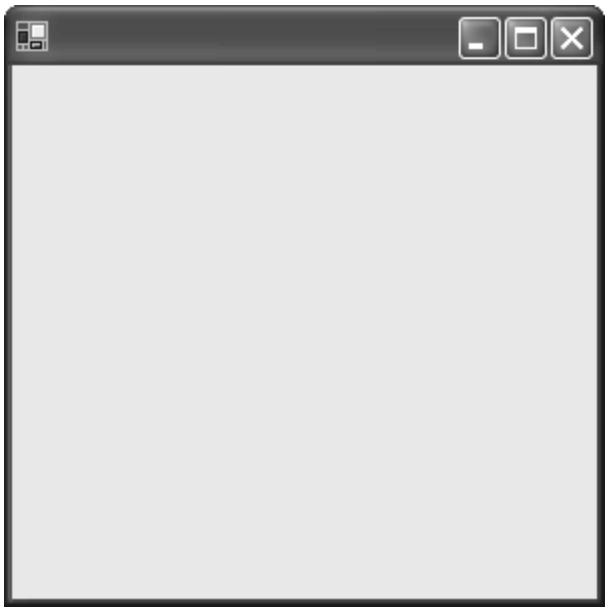


Figure 14-4: Not bad for 8 lines of code

The base class **Form** contains more than 100 public and protected properties, a similar number of methods, and more than 70 events and corresponding event-raising methods. But it doesn't stop there; **Form** is a subtype of a class called **Control** (not a direct subtype, it's actually **Form** : **ContainerControl** : **ScrollableControl** : **Control**). Instances of **Control** have a property called **Controls** which contains a collection of other controls. This structure, an example of the *Composite* design pattern, allows everything from simple buttons to the most complex user-interfaces to be treated uniformly by programmers and development tools such as the visual builder tool in Visual Studio .NET.

GUI architectures

Architectures were presented in Chapter 9 as an "overall ordering principle" of a system or subsystem. While the **Controls** property of **Control** is an ordering principle for the static structure of the widgets in a Windows Forms application, Windows Forms does not dictate an ordering principle for associating these widgets with particular events and program logic. Several architectures are possible with Windows Forms, and each has its strengths and weaknesses.

It's important to have a consistent UI architecture because, as Larry Constantine and Lucy Lockwood point out, while the UI is just one, perhaps uninteresting, part of the system to the programmer, to the end user, the UI *is* the program. The UI is the entry point for the vast majority of change requests, so you'd better make it easy to change the UI without changing the logical behavior of the program. Decoupling the presentation layer from the business layer is a fundamental part of professional development.

Using the Visual Designer

Open Visual Studio .NET, bring up the New Project wizard, and create a Windows Application called FormControlEvent. The wizard will generate some source code and present a "Design View" presentation of a blank form. Drag and drop a button and label onto the form. You should see something like this:

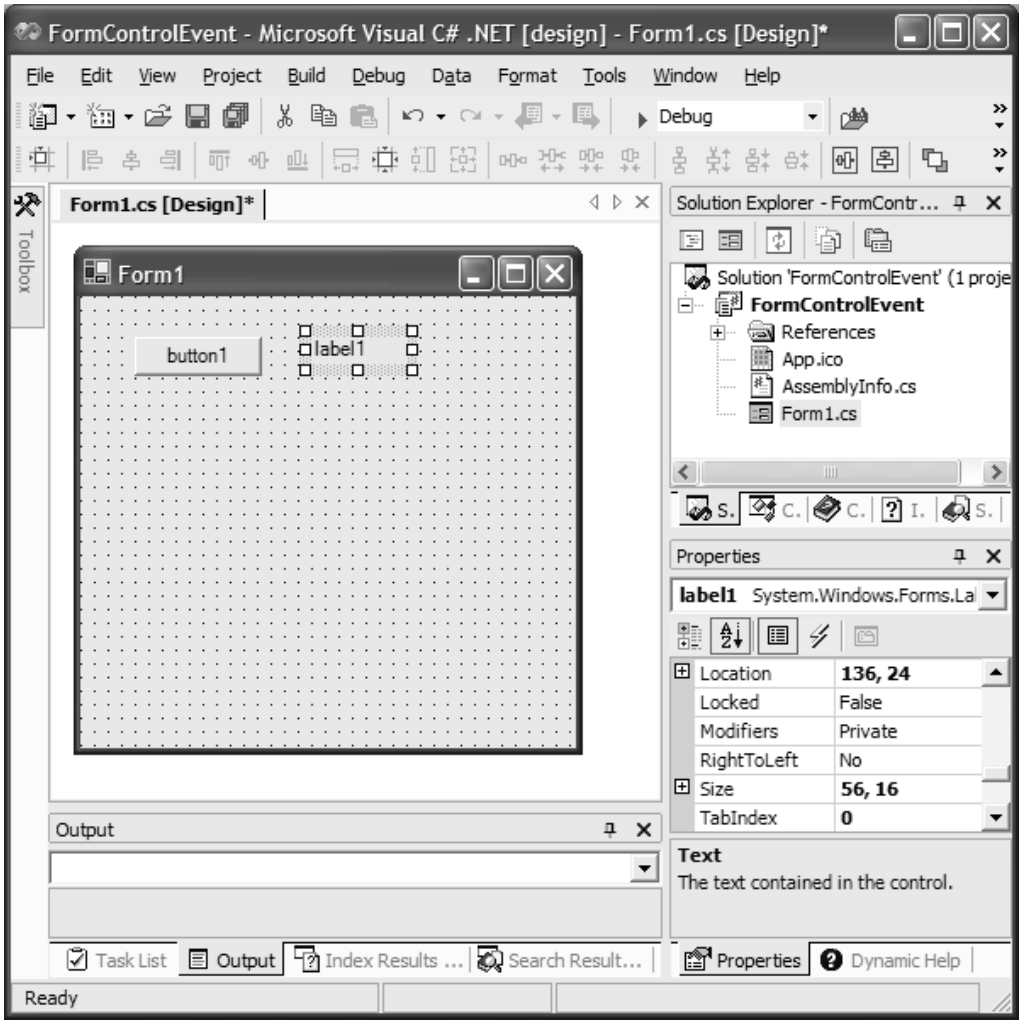


Figure 14-5: Visual Studio.NET makes C# programming as easy as Visual Basic

In the designer, double-click the button. Visual Studio will switch to a code-editing view, with the cursor inside a method called **button1_Click( )**. Add the line;

```
label1.Text = "Clicked";
```

The resulting program should look a lot like this:

```
//:c14:FormControlEvent.cs
//Designer-generated Form-Control-Event architecture
using System;
```

```
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace FormControlEvent{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form {
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Button button1;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container
    components = null;

    public Form1(){
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      // TODO: Add any constructor code after
      // InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose(bool disposing ){
      if ( disposing ) {
        if (components != null) {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
```

```
    }

#region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support
    /// - do not modify the contents of this method
    /// with the code editor.
    /// </summary>
    private void InitializeComponent(){
      this.label1 =
        new System.Windows.Forms.Label();
      this.button1 =
        new System.Windows.Forms.Button();
      this.SuspendLayout();
      //
      // label1
      //
      this.label1.Location =
        new System.Drawing.Point(136, 24);
      this.label1.Name = "label1";
      this.label1.Size =
        new System.Drawing.Size(56, 16);
      this.label1.TabIndex = 0;
      this.label1.Text = "label1";
      //
      // button1
      //
      this.button1.Location =
        new System.Drawing.Point(32, 24);
      this.button1.Name = "button1";
      this.button1.TabIndex = 1;
      this.button1.Text = "button1";
      this.button1.Click +=
        new System.EventHandler(this.button1_Click);
      //
      // Form1
      //
      this.AutoScaleBaseSize =
        new System.Drawing.Size(5, 13);
      this.ClientSize =
        new System.Drawing.Size(292, 266);
```

```
      this.Controls.AddRange(
        new System.Windows.Forms.Control[]{
          this.button1, this.label1});
      this.Name = "Form1";
      this.Text = "Form1";
      this.ResumeLayout(false);
    }
#endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main() {
      Application.Run(new Form1());
    }

    private void button1_Click(
      object sender, System.EventArgs e) {
      label1.Text = "Clicked";
    }
  }
}///:~
```

The first interesting detail is the **#region** - **#endregion** statements. These preprocessing directives (see Chapter 4) delineate a code section that Visual Studio .NET may collapse in its "outlining" mode; indeed, when you first switch to code view, this area of the code was probably somewhat hidden from view. While it's generally a good idea to heed the warning about not editing Designer-generated code, the code is well worth taking a closer look at.

The label and button that we dragged onto the form are initialized as new objects from the System.Windows.Forms namespace. The call to **SuspendLayout( )** indicates that a series of manipulations are coming and that each individual one should not trigger the potentially expensive layout calculation on the **Control** and all of its sub-**Controls**. Some of the basic properties for each control are then set:

- **Location** specifies the point where the upper-left corner of the control is relative to the upper-left corner of the containing control or, if the **Control** is a **Form Location** is the screen coordinates of the upper-left corner (including the **Form**'s border if it has one, as most do). This is a

value that you can freely manipulate without worrying about the ?resizing behavior of the **Form**.

- **Size** is measured in pixels. Like **Location**, this property returns a value, not a reference, so to manipulate the **Control**, you must assign any change to the property to have any effect:

```
Size s = myControl.Size;
s.Width += 10; //not a reference, no change to control
myControl.Size = s; //Now control will change
```

- **TabIndex** specifies the order in which a control is activated when the user presses the Tab key.
- **Text** is displayed in various ways, depending upon the **Control**'s type. The **Text** of the form, for instance, is displayed as the Window's title, while the **Button** and **Label** have other properties such as **TextAlign** and **Font** to fine-tune their appearance (**Form** has a **Font** property, too, but it just sets the default font for its subcontrols; it does not change the way the title of the **Form** is displayed). The **Name** property corresponds to the named variable that represents the control and is necessary for the visual designer to work; don't manually change this.

The final part of the block of code associated with **button1** reads:

```
this.button1.Click +=
   new System.EventHandler(this.button1_Click);
```

From our previous discussion of multicast delegates, this should be fairly easy to interpret: **Button** has an event property **Click** which specifies a multicast delegate of type **EventHandler**. The method **this.button1_Click( )** is being added as a multicast listener.

At the bottom of the **InitializeComponent** method, additional properties are set for the **Form1** itself. **AutoScaleBaseSize** specifies how the **Form** will react if the **Form**'s font is changed to a different size (as can happen by default in Windows). **ClientSize** is the area of the **Control** in which other **Control**'s can be placed; in the case of a window, that excludes the title bar and border, scrollbars are also not part of the *client area*.

The method **Controls.AddRange( )** places an array of **Control**s in the containing **Control**. There is also an **Add( )** method which takes a single control, but the visual designer always uses **AddRange( )**.

Finally, **ResumeLayout( ),** the complement to **SuspendLayout( )**, reactivates layout behavior. The visual designer passes a **false** parameter, indicating that it's not necessary to force an immediate relayout of the **Control**.

The **Main( )** method is prepended with an **[STAThread]** attribute, which sets the threading model to "single-threaded apartment." We'll discuss this attribute briefly in Chapter 15.

The last method is the **private** method **button1_Click( )**, which was attached to **button1**'s **Click** event property in the **InitializeComponent( )** method. In this method we directly manipulate the **Text** property of the **label1** control.

Some obvious observations about the output of the visual designer: It works with code that is both readable and (despite the warning) editable, the visual designer works within the monolithic **InitializeComponent( )** except that it creates event-handler methods that are in the same **Control** class being defined, and the code isn't "tricky" other than the **[STAThread]** attribute and the **Dispose( )** method (a method which is not necessary unless the **Control** or one of its subcontrols contains non-managed resources, as discussed in Chapter 5).

Less obviously, taken together, the visual designer *does* implicitly impose "an overall ordering principle" to the system. The visual designer constructs applications that have a statically structured GUI, individual identity for controls and handlers, and localized event-handling.

The problem with this architecture, as experienced Visual Basic programmers can attest, is that people can be fooled into thinking that the designer is "doing the object orientation" for them and event-handling routines become monolithic procedural code chunks. This can also lead to people placing the domain logic directly in handlers, thus foregoing the whole concept of decoupling UI logic from domain logic.

This is a prime example of where sample code such as is shown in articles or this book is misleading. Authors and teachers will generally place domain logic inline with a control event in order to save space and simplify the explanation (as we did with the **label1.Text = "Clicked"** code). However, in professional development, the structure of pretty much any designer-generated event handler should probably be:

```
private void someControl_Click(
    object sender, System.EventArgs e) {
       someDomainObject.SomeLogicalEvent();
}
```

This structure separates the concept of the **Control** and GUI events from domain objects and logical events and a GUI that uses this structure will be able to change its domain logic without worrying about the display details.

Unfortunately, alerting domain objects to GUI events is only half the battle, the GUI must somehow reflect changes in the state of domain objects. This challenge has several different solutions.

Form-Event-Control

The first GUI architecture we'll discuss could be called "Form-Event-Control." The FEC architecture uses a unified event-driven model: GUI objects create GUI events that trigger domain logic that create domain events that trigger GUI logic. This is done by creating domain event properties and having controls subscribe to them, as this example shows:

```
//:c14:FECDomain.cs
using System;
using System.Text.RegularExpressions;
using System;
delegate void StringSplitHandler(
  object src, SplitStringArgs args);

class SplitStringArgs : EventArgs {
  private SplitStringArgs(){}
  public SplitStringArgs(string[] strings){
    this.strings = strings;
  }

  string[] strings;
  public string[] Strings{
    get { return strings;}
    set { strings = value;}
  }
}

class DomainSplitter {
  Regex re = new Regex("\\s+");
  string[] substrings;

  public event StringSplitHandler StringsSplit;

  public void SplitString(string inStr){
    substrings = re.Split(inStr);
    StringsSplit(
      this, new SplitStringArgs(substrings));
```

```
  }
}///:~
```

This is our domain object, which splits a string into its substrings with the **Regex.Split( )** method. When this happens, the **DomainSplitter** raises a **StringsSplit** event with the newly created substrings as an argument to its **SplitStringArgs**. Now to create a Windows Form that interacts with this domain object:

```
//:c14:FECDomain2.cs
//Compile with csc FECDomain FECDomain2
using System;
using System.Drawing;
using System.Windows.Forms;

class FECDomain : Form {
  TextBox tb = new TextBox();
  Button b = new Button();
  Label[] labels;
  DomainSplitter domainObject = new DomainSplitter();

  FECDomain(){
    tb.Location = new Point(10, 10);
    tb.Text = "The quick brown fox";
    b.Location = new Point(150, 10);
    b.Text = "Split text";

    b.Click += new EventHandler(this.GUIEvent);
    domainObject.StringsSplit +=
      new StringSplitHandler(this.DomainEvent);

    this.Text = "Form-Event-Control";
    this.Controls.Add(tb);
    this.Controls.Add(b);
  }

  void GUIEvent(object src, EventArgs args){
    domainObject.SplitString(tb.Text);
  }

  void DomainEvent(object src, SplitStringArgs args){
    string[] strings = args.Strings;
```

```
    if (labels != null) {
      foreach(Label l in labels){
        this.Controls.Remove(l);
      }
    }
    labels = new Label[strings.Length];
    int row = 40;
    for (int i = 0; i < labels.Length; i++) {
      labels[i] = new Label();
      labels[i].Text = strings[i];
      labels[i].Location = new Point(100, row);
      row += 20;
    }
    this.Controls.AddRange(labels);
  }

  public static void Main(){
    Application.Run(new FECDomain());
  }
}///:~
```

Obviously, we didn't use Visual Studio's designer to build this form but have reverted to working directly from within a code editor. Our **FECDomain** form contains a text box, a button, an array of Label controls, and a reference to **DomainSplitter**.

The first part of the **FEDomain( )** constructor specifies the location and text of the text box and button. We then specify two delegates: **GUIEvent** is a delegate of type **EventHandler** and is attached to the button's **Click** event property and **DomainEvent** is of type **StringSplitHandler** and is attached to the **DomainSplitter**'s **StringSplit** event. The final part of the constructor adds the textbox and button to the form.

When the button is pressed, the **Click** delegate invokes the **GUIEvent( )** method, which passes the text of the textbox to the **domainObject.SplitString( )** logical event. This in turn will raise a **StringSplit** event that calls back to the **DomainEvent( )** method.

The **DomainEvent( )** method creates and displays a label for each of the individual strings. The first time **DomainEvent( )** is called, the **labels** array will be null because we do not initialize it in the constructor. If, though, **labels** is not null, we remove the existing labels from the **Controls** collection. We initialize the **labels** array to be able to hold a sufficient number of references and

then initialize individual labels with the appropriate **string** and a new position. Once all the **labels** are created, **Controls.AddRange( )** adds them to the **FECDomain**'s client area.

The FEC architecture is vulnerable to the recursive loops problems discussed previously. If a domain event triggers a GUI handler which in turn activates the relevant domain event, the system will recurse and crash (when dealing with GUIs, the crash exception typically involves starvation of some Windows resource before the stack overflows). However, FEC is very straightforward – although in the tiny programs that illustrate a book it is more complex than just putting domain logic directly into a GUI event handler, in practice it will very likely be less complex and provides for a very clean and understandable separation of GUI and domain logic.

Presentation-Abstraction-Control

An alternative GUI architecture to FEC proposes that the whole concept of separating domain logic from **Control**s is overemphasized. In this view, flexibility is achieved by encapsulating all the display, control, and domain logic associated with a relatively fine-grained abstraction. Groups of these self-contained components are combined to build coarser-grained abstractions (with correspondingly more complex displays, perhaps panels and entire forms). These coarser-grained abstractions are gathered together to make programs.

In the PAC architecture, the lowest-level objects are likely to be subtypes of specific controls; for instance, a **Button** that encapsulates a bit of domain logic relating to a trigger or switch. Mid-level objects may descend from **UserControl** (essentially, an interface-less **Control**) and would encapsulate discrete chunks of business logic. Higher-level objects would likely descend from **Form** and are likely to encapsulate all the logic associated with a particular scenario or use-case.

In this example, we have a type descended from **Button** that knows whether it is on or off and a type descended from **Panel** that contains these **TwoState** buttons and knows if all the **TwoState**s within it are in state "On":

```
//:c14:PAC.cs
//Presentation-Abstraction-Control
using System.Drawing;
using System.Windows.Forms;
using System;

class TwoState : Button {
  static int instanceCounter = 0;
```

```
  int id = instanceCounter++;

  internal TwoState(){
    this.Text = State;
    System.EventHandler hndlr =
    new System.EventHandler(buttonClick);
    this.Click += hndlr;
  }

  bool state = true;
  public string State{
    get {
      return(state == true) ? "On" : "Off";
    }
    set{
      state = (value == "On") ? true : false;
      OnStateChanged();
    }
  }

  private void buttonClick(
    object sender, System.EventArgs e){
    changeState();
  }

  public void changeState(){
    state = !state;
    OnStateChanged();
  }

  public void OnStateChanged(){
    Console.WriteLine(
      "TwoState id " + id + " state changed");
    this.Text = State;
  }
}

class ChristmasTree : Panel {
  bool allOn;
  internal bool AllOn{
    get { return allOn;}
```

```
}

public ChristmasTree(){
  TwoState ts = new TwoState();
  ts.Location = new Point(10, 10);
  TwoState ts2 = new TwoState();
  ts2.Location = new Point(120, 10);
  Add(ts);
  Add(ts2);
  BackColor = Color.Green;
}

public void Add(TwoState c){
  Controls.Add(c);
  c.Click += new EventHandler(
    this.TwoClickChanged);
}

public void AddRange(TwoState[] ca){
  foreach(TwoState ts in ca){
    ts.Click += new EventHandler(
      this.TwoClickChanged);
  }
  Controls.AddRange(ca);
}

public void TwoClickChanged(
  Object src, EventArgs a){
  allOn = true;
  foreach(Control c in Controls){
    TwoState ts = c as TwoState;
    if (ts.State != "On") {
      allOn = false;
    }
  }
  if (allOn) {
    BackColor = Color.Green;
  } else {
    BackColor = Color.Red;
  }
}
```

```
}

class PACForm : Form {
  ChristmasTree p1 = new ChristmasTree();
  ChristmasTree p2 = new ChristmasTree();

  public PACForm(){
    ClientSize = new Size(450, 200);
    Text = "Events & Models";

    p1.Location = new Point(10,10);
    p2.Location = new Point(200, 10);
    Controls.Add(p1);
    Controls.Add(p2);
  }

  static void Main(){
    Application.Run(new PACForm());
  }
}///:~
```

When run, if you set both the buttons within an individual **ChristmasTree** panel to "On," the **ChristmasTree**'s background color will become green, otherwise, the background color will be red. The **PACForm** knows nothing about the **TwoState**s within the **ChristmasTree**. We could (and indeed it would probably be logical) change **TwoState** from descending from **Button** to descending from **Checkbox** and **TwoState.State** from a **string** to a **bool** and it would make no difference to the **PACForm** that contains the two instances of **ChristmasTree**.

Presentation-Abstraction-Control is often the best architecture for working with .NET. Its killer advantage is that it provides an encapsulated component. A component is a software module that can be deployed and composed into other components *without source-code modification*. Visual Basic programmers have enjoyed the benefits of software components for more than a decade, with thousands of third-party components available. Visual Studio .NET ships with a few components that are at a higher level than just encapsulating standard controls, for instance, components which encapsulate ADO and another which encapsulates the Crystal Reports tools.

PAC components need not really be written as a single class; rather, a single **Control** class may provide a single **public** view of a more complex namespace

whose members are not visible to the outside world. This is called the *Façade* design pattern.

The problem with PAC is that it's hard work to create a decent component. Our **ChristmasTree** component is horrible – it doesn't automatically place the internal **TwoState**s reasonably, it doesn't resize to fit new **TwoState**s, it doesn't expose both logical and GUI events and properties… The list goes on and on. Reuse is the great unfulfilled promise of object orientation: "Drop a control on the form, set a couple of properties, and boom! You've got a payroll system." But the reality of development is that at least 90% of your time is absolutely controlled by the pressing issues of the current development cycle and there is little or no time to spend on details not related to the task at hand. Plus, it's difficult enough to create an effective GUI when you have direct access to your end-users; creating an effective GUI component that will be appropriate in situations that haven't yet arisen is almost impossible.

Nevertheless, Visual Studio .NET makes it so easy to create a reusable component (just compile your component to a .DLL and you can add it to the Toolbar!) that you should always at least consider PAC for your GUI architecture.

Model-View-Controller

Model-View-Controller, commonly referred to simply as "MVC," was the first widely known architectural pattern for decoupling the graphical user interface from underlying application logic. Unfortunately, many people confuse MVC with any architecture that separates presentation logic from domain logic. So when someone starts talking about MVC, it's wise to allow for quite a bit of imprecision.

In MVC, the Model encapsulates the system's logical behavior and state, the View requests and reflects that state on the display, and the Controller interprets low-level inputs such as mouse and keyboard strokes, resulting in commands to either the Model or the View.

MVC trades off a lot of static structure — the definition of objects for each of the various responsibilities – for the advantage of being able to independently vary the view and the controller. This is not much of an advantage in Windows programs, where the view is always a bitmapped two-dimensional display and the controller is always a combination of a keyboard and a mouse-like pointing device. However, USB's widespread support has already led to interesting new controllers and the not-so-distant future will bring both voice and gesture control and "hybrid reality" displays to seamlessly integrate computer-generated data into real vision (e.g., glasses that superimpose arrows and labels on reality). If you happen to be lucky enough to be working with such advanced technologies,

MVC may be just the thing. Even if not, it's worth discussing briefly as an example of decoupling GUI concerns taken to the logical extreme. In our example, our domain state is simply an array of Boolean values; we want the display to show these values as buttons and display, in the title bar, whether all the values are true or whether some are false:

```
//:c14:MVC.cs
using System;
using System.Windows.Forms;
using System.Drawing;

class Model {
  internal bool[] allBools;

  internal Model(){
    Random r = new Random();
    int iBools = 2 + r.Next(3);
    allBools = new bool[iBools];
    for (int i = 0; i < iBools; i++) {
      allBools[i]  = r.NextDouble() > 0.5;
    }
  }
}

class View : Form {
  Model model;
  Button[] buttons;

  internal View(Model m, Controller c){
    this.model = m;

    int buttonCount = m.allBools.Length;
    buttons = new Button[buttonCount];
    ClientSize =
      new Size(300, 50 + buttonCount * 50);
    for (int i = 0; i < buttonCount; i++) {
      buttons[i] = new Button();
      buttons[i].Location =
        new Point(10, 5 + i * 50);
      c.MatchControlToModel(buttons[i], i);
      buttons[i].Click +=
```

```
          new EventHandler(c.ClickHandler);
      }
      ReflectModel();
      Controls.AddRange(buttons);
    }

    internal void ReflectModel(){
      bool allAreTrue = true;
      for (int i = 0; i < model.allBools.Length; i++) {
        buttons[i].Text = model.allBools[i].ToString();
        if (model.allBools[i] == false) {
          allAreTrue = false;
        }
      }
      if (allAreTrue) {
        Text = "All are true";
      } else {
        Text = "Some are not true";
      }
    }
}

class Controller {
  Model model;
  View view;

  Control[] viewComponents;

  Controller(Model m){
    model = m;
    viewComponents = new Control[m.allBools.Length];
  }

  internal void MatchControlToModel
    (Button b, int index){
    viewComponents[index] = b;
  }

  internal void AttachView(View v){
    this.view = v;
  }
```

```
  internal void ClickHandler
    (Object src, EventArgs ea){
    //Modify model in response to input
    int modelIndex =
      Array.IndexOf(viewComponents, src);
    model.allBools[modelIndex] =
      !model.allBools[modelIndex];
    //Have view reflect model
    view.ReflectModel();
  }

  public static void Main(){
    Model m = new Model();
    Controller c = new Controller(m);
    View v = new View(m, c);
    c.AttachView(v);
    Application.Run(v);
  }
}///:~
```

The **Model** class has an array of **bools** that are randomly set in the **Model** constructor. For demonstration purposes, we're exposing the array directly, but in real life the state of the **Model** would be reflected in its entire gamut of properties.

The **View** object is a **Form** that contains a reference to a **Model** and its role is simply to reflect the state of that **Model**. The **View( )** constructor lays out how this particular view is going to do that – it determines how many **bool**s are in the **Model**'s **allBools** array and initializes a **Button[]** array of the same size. For each **bool** in **allBools**, it creates a corresponding **Button,** and associates this particular aspect of the **Model**'s state (the index of this particular **bool**) with this particular aspect of the **View** (this particular **Button**). It does this by calling the **Controller**'s **MatchControlToModel( )** method and by adding to the **Button**'s **Click** event property a reference to the **Controller**'s **ClickHandler( )**. Once the **View** has initialized its **Controls,** it calls its own **ReflectModel( )** method.

View's **ReflectModel( )** method iterates over all the **bool**s in **Model**, setting the text of the corresponding button to the value of the Boolean**.** If all are **true**, the title of the **Form** is changed to "All are true," otherwise, it declares that "Some are false."

The **Controller** object ultimately needs references to both the **Model** and to the **View**, and the **View** needs a reference to both the **Model** and the **Controller**. This leads to a little bit of complexity in the initialization shown in the **Main( )** method; the **Model( )** is created with no references to anything (the **Model** is acted upon by the **Controller** and reflected by the **View**, the **Model** itself never needs to call methods or properties in those objects). The **Controller** is then created with a reference to the **Model**. The **Controller( )** constructor simply stores a reference to the **Model** and initializes an array of **Control**s to sufficient size to reflect the size of the **Model.allBools** array. At this point, the **Controller.View** reference is still null, since the **View** has not yet been initialized.

Back in the **Main( )** method, the just-created **Controller** is passed to the **View( )** constructor, which initializes the **View** as described previously. Once the **View** is initialized, the **Controller**'s **AttachView( )** method sets the reference to the **View**, completing the **Controller**'s initialization. (You could as easily do the opposite, creating a **View** with just a reference to the **Model**, creating a **Controller** with a reference to the **View** and the **Model**, and then finish the **View**'s initialization with an **AttachController( )** method.)

During the **View'**s constructor, it called the **Controller**'s **MatchControlToModel( )** method, which we can now see simply stores a reference to a **Button** in the **viewComponents[ ]** array.

The **Controller** is responsible for interpreting events and causing updates to the **Model** and **View** as appropriate. **ClickHandler( )** is called by the various **Button**s in the **View** when they are clicked. The originating **Control** is referenced in the **src** method argument, and because the index in the **viewComponents[ ]** array was defined to correspond to the index of the **Model**'s **allBools[ ]** array, we can learn what aspect of the **Model**'s state we wish to update by using the static method **Array.IndexOf( )**. We change the Boolean to its opposite using the ! operator and then, having changed the **Model**'s state, we call **View**'s **ReflectModel( )** method to keep everything in synchrony.

The clear delineation of duties in MVC is appealing – the **View** passively reflects the **Model**, the **Controller** mediates updates**,** and the **Model** is responsible only for itself. You can have many **View** classes that reflect the same **Model** (say, one showing a graph of values, the other showing a list) and dynamically switch between them. However, the structural complexity of MVC is a considerable burden and is difficult to "integrate" with the Visual Designer tool.

Layout

Now that we've discussed the various architectural options that should be of major import in any real GUI design discussion, we're going to move back towards the expedient "Do as we say, not as we do" mode of combining logic, event control, and visual display in the sample programs. It would simply consume too much space to separate domain logic into separate classes when, usually, our example programs are doing nothing but writing out simple lines of text to the console or demonstrating the basics of some simple widget.

Another area where the sample programs differ markedly from professional code is in the layout of **Control**s. So far, we have used the **Location** property of a **Control,** which determines the upper-left corner of **this Control** in the client area of its containing **Control** (or, in the case of a **Form**, the Windows display coordinates of its upper-left corner).

More frequently, you will use the **Dock** and **Anchor** properties of a **Control** to locate a **Control** relative to one or more *edges* of the container in which it resides. These properties allow you to create **Control**s which properly resize themselves in response to changes in windows size.

In our examples so far, resizing the containing Form doesn't change the **Control** positions. That is because by default, **Control**s have an **Anchor** property set to the **AnchorStyles** values **Top** and **Left** (**AnchorStyles** are bitwise combinable). In this example, a button moves relative to the opposite corner:

```
//:c14:AnchorValues.cs
using System.Windows.Forms;
using System.Drawing;

class AnchorValues: Form {
  AnchorValues(){
    Button b = new Button();
    b.Location = new Point(10, 10);
    b.Anchor =
    AnchorStyles.Right | AnchorStyles.Bottom;
    Controls.Add(b);
  }

  public static void Main(){
    Application.Run(new AnchorValues());
  }
```

```
}///:~
```

If you combine opposite **AnchorStyles** (Left and Right, Top and Bottom) the **Control** will resize. If you specify **AnchorStyles**.**None**, the control will move half the distance of the containing area's change in size. This example shows these two types of behavior:

```
//:c14:AnchorResizing.cs
using System.Windows.Forms;
using System.Drawing;

class AnchorResizing: Form {
  AnchorResizing(){
    Button b = new Button();
    b.Location = new Point(10, 10);
    b.Anchor =
      AnchorStyles.Left
      | AnchorStyles.Right
      | AnchorStyles.Top;
    b.Text = "Left | Right | Top";
    Controls.Add(b);

    Button b2 = new Button();
    b2.Location = new Point(100, 10);
    b2.Anchor = AnchorStyles.None;
    b2.Text = "Not anchored";
    Controls.Add(b2);
  }

  public static void Main(){
    Application.Run(new AnchorResizing());
  }
}///:~
```

If you run this example and manipulate the screen, you'll see two undesirable behaviors: **b** can obscure **b2**, and **b2** can float off the page. Windows Forms' layout behavior trades off troublesome behavior like this for its straightforward model. An alternative mechanism based on cells, such as that used in HTML or some of Java's **LayoutManagers**, may be more robust in avoiding these types of trouble, but anyone who's tried to get a complex cell-based UI to resize the way they wish is likely to agree with Windows Forms' philosophy!

A property complementary to **Anchor** is **Dock**. The **Dock** property moves the control flush against the specified edge of its container, and resizes the control to be the same size as that edge. If more than one control in a client area is set to **Dock** to the same edge, the controls will layout side-by-side in the *reverse of the order* in which they were added to the containing **Controls** array (their *reverse z-order*). The **Dock** property overrides the **Location** value. In this example, two buttons are created and docked to the left side of their containing **Form**.

```
//:c14:Dock.cs1
using System.Windows.Forms;
using System.Drawing;

class Dock: Form {
  Dock(){
    Button b1 = new Button();
    b1.Dock = DockStyle.Left;
    b1.Text = "Button 1";
    Controls.Add(b1);

    Button b2 = new Button();
    b2.Dock = DockStyle.Left;
    b2.Text = "Button2";
    Controls.Add(b2);
  }

  public static void Main(){
    Application.Run(new Dock());
  }
}///:~
```

When you run this, you'll see that **b** appears *to the right* of **b2** because it was added to **Dock**'s **Controls** *before* **b2.**

DockStyle.Fill specifies that the **Control** will expand to fill the client area from the center to the limits allowed by other **Dock**ed **Controls**. **DockStyle.Fill** will cover non-**Dock**ed **Controls** that have a lower z-order, as this example shows:

```
//:c14:DockFill.cs
using System.Windows.Forms;
using System.Drawing;

class DockFill: Form {
  DockFill(){
```

```
    //Lower z-order
    Button visible = new Button();
    visible.Text = "Visible";
    visible.Location = new Point(10, 10);
    Controls.Add(visible);

    //Will cover "Invisible"
    Button docked = new Button();
    docked.Text = "Docked";
    docked.Dock = DockStyle.Fill;
    Controls.Add(docked);

    //Higher z-order, gonna' be invisible
    Button invisible = new Button();
    invisible.Text = "Invisible";
    invisible.Location = new Point(100, 100);
    Controls.Add(invisible);
  }

  public static void Main(){
    Application.Run(new DockFill());
  }
}///:~
```

Developing complex layouts that lay themselves out properly when resized is a challenge for any system. Windows Forms' straightforward model of containment, **Location, Anchor**, and **Dock** is especially suited for the PAC GUI architecture described previously. Rather than trying to create a monolithic chunk of logic that attempts to resize and relocate the hundreds or dozens of widgets that might comprise a complex UI, the PAC architecture would encapsulate the logic within individual custom controls.

Non-code resources

Windows Forms wouldn't be much of a graphical user interface library if it did not support graphics and other media. But while it's easy to specify a **Button**'s look and feel with only a few lines of code, images are inherently dependent on binary data storage.

It's not surprising that you can load an image into a Windows Form by using a **Stream** or a filename, as this example demonstrates:

```
//:c14:SimplePicture.cs
```

```
//Loading images from file system

using System;
using System.IO;
using System.Windows.Forms;
using System.Drawing;

class SimplePicture : Form {
  public static void Main(string[] args){
    SimplePicture sp = new SimplePicture(args[0]);
    Application.Run(sp);
  }

  SimplePicture(string fName){
    PictureBox pb = new PictureBox();
    pb.Image = Image.FromFile(fName);
    pb.Dock = DockStyle.Fill;
    pb.SizeMode = PictureBoxSizeMode.StretchImage;
    Controls.Add(pb);

    int imgWidth = pb.Image.Width;
    int imgHeight = pb.Image.Height;

    this.ClientSize =
    new Size(imgWidth, imgHeight);
  }
}///:~
```

The **Main( )** method takes the first command-line argument as a path to an image (for instance: "SimplePicture c:\windows\clouds.bmp") and passes that path to the **SimplePicture( )** constructor. The most common **Control** used to display a bitmap is the **PictureBox** control, which has an **Image** property. The static method **Image.FromFile( )** generates an **Image** from the given path (there is also an **Image.FromStream( )** static method which provides general access to all the possible sources of image data).

The **PictureBox**'s **Dock** property is set to **Dockstyle.Fill** and the **ClientSize** of the form is set to the size of the **Image**. When you run this program, the **SimplePicture** form will start at the same size of the image. Because **pb.SizeMode** was set to **StretchImage**, however, you can resize the form and the image will stretch or shrink appropriately. Alternate **PictureBoxSizeMode** values are **Normal** (which clips the **Image** to the **PictureBox**'s size),

AutoSize (which resizes the **PictureBox** to accommodate the **Image**'s size), and **CenterImage.**

Loading resources from external files is certainly appropriate in many circumstances, especially with isolated storage (#ref#), which gives you a per-user, consistent virtual file system. However, real applications which are intended for international consumption require many resources localized to the current culture – labels, menu names, and icons may all have to change. The .NET Framework provides a standard model for efficiently storing such resources and loading them. Momentarily putting aside the question of how such resources are created, retrieving them is the work of the **ResourceManager** class. This example switches localized labels indicating "man" and "woman."

```
//:c14:International.cs
using System;
using System.Drawing;
using System.Resources;
using System.Globalization;
using System.Windows.Forms;

class International : Form {
  ResourceManager rm;
  Label man;
  Label woman;

  public International(){
    rm = new ResourceManager(typeof(International));

    RadioButton eng = new RadioButton();
    eng.Checked = true;
    eng.Location = new Point(10, 10);
    eng.Text = "American";
    eng.CheckedChanged +=
      new EventHandler(LoadUSResources);

    RadioButton swa = new RadioButton();
    swa.Location = new Point(10, 30);
    swa.Text = "Swahili";
    swa.CheckedChanged +=
      new EventHandler(LoadSwahiliResources);

    man = new Label();
```

```
    man.Location = new Point(10, 60);
    man.Text = "Man";

    woman = new Label();
    woman.Location = new Point(10, 90);
    woman.Text = "Woman";

    Controls.AddRange(new Control[]{
      eng, swa, man, woman});

    Text = "International";
  }

  public void LoadUSResources
    (Object src, EventArgs a){
    if ( ((RadioButton)src).Checked == true) {
      ResourceSet rs =
        rm.GetResourceSet(
          new CultureInfo("en-US"), true, true);
      SetLabels(rs);
    }
  }

  public void LoadSwahiliResources
    (Object src, EventArgs a){
    if (((RadioButton)src).Checked == true) {
      ResourceSet rs =
      rm.GetResourceSet(
        new CultureInfo("sw"), true, true);
      SetLabels(rs);
    }
  }

  private void SetLabels(ResourceSet rs){
    man.Text = rs.GetString("Man");
    woman.Text = rs.GetString("Woman");
  }

  public static void Main(){
    Application.Run(new International());
  }
```

```
}///:~
```

Here, we wish to create an application which uses labels in a local culture (a culture is more specific than a language; for instance, there is a distinction between the culture of the United States and the culture of the United Kingdom). The basic references we'll need are to **ResourceManager rm**, which we'll load to be culture-specific, and two labels for the words "Man" and "Woman."

The first line of the **International** constructor initializes **rm** to be a resource manager for the specified type. A **ResourceManager** is associated with a specific type because .NET uses a "hub and spoke" model for managing resources. The "hub" is the assembly that contains the code of a specific type. The "spokes" are zero or more *satellite assemblies* that contain the resources for specific cultures, and the **ResourceManager** is the link that associates a type (a "hub") with its "spokes."

International( ) then shows the use of radio buttons in Windows Forms. The model is simple: All radio buttons within a container are mutually exclusive. To make multiple sets of radio buttons within a single form, you can use a **GroupBox** or **Panel**. **International** has two **RadioButton**s, **eng** has its **Checked** property set to **true**, and, when that property changes, the **LoadEnglishResources** method will be called. **RadioButton swa** is similarly configured to call **LoadSwahiliResources( )** and is not initially checked. By default, the **man** and **woman** labels are set to a hard-coded value.

The **Load*xxx*Resources( )** methods are similar; they check if their source **RadioButton** is checked (since they are handling the **CheckedChange** event, the methods will be called when their source becomes unchecked as well). If their source *is* set, the **ResourceManager** loads one of the "spoke" **ResourceSet** objects. The **ResourceSet** is associated with a particular **CultureInfo** instance, which is initialized with a *language tag* **string** compliant with IETF RFC 1766 (you can find a list of standard codes in the .NET Framework documentation and read the RFC at *http://www.ietf.org/rfc/rfc1766.txt*). The **GetResourceSet( )** method also takes two **bools**, the first specifying if the **ResourceSet** should be loaded if it does not yet exist in memory, and the second specifying if the **ResourceManager** should try to load "parents" of the culture if the specified **CultureInfo** does not work; both of these **bool**s will almost always be **true**.

Once the **ResourceSet** is retrieved, it is used as a parameter to the **SetLabels( )** method. **SetLabels( )** uses **ResourceSet.GetString( )** to retrieve the appropriate culture-specific **string** for the specified key and sets the associated **Label.Text**. **ResourceSet**'s other major method is **GetObject( )** which can be used to retrieve any type of resource.

We've not yet created the satellite assemblies which will serve as the "spokes" to our **International** "hub," but it is interesting to run the program in this state. If you run the above code and click the "Swahili" radio button, you will see this dialog:


International
An unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will be shut down immediately.
Could not find any resources appropriate for the specified culture (or the neutral culture) in the given assembly. Make sure "International.resources" was correctly embedded or linked into assembly "International".
baseName: International locationInfo: International resource file name: International.resources assembly: International, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null.
Details
Continue
Quit

Figure 14-6: A detailed exception handling dialog from Microsoft

This is not a dialog you'd ever want an end-user to see and a real application's exception handling would hide it, but it's an interesting example of the kind of behavior that you could potentially include in your own components to aid 3rd party developers during debugging.

Creating satellite assemblies

For your satellite assembly to work, you must follow naming (including capitalization) and directory conventions. First, you will create a new subdirectory for each culture you wish to support and named with the culture's language tag. If you compiled **International.exe** in the directory c:\tic\chap14, you will create c:\tic\chap14\sw and c:\tic\chap14\en-US.

In the \sw subdirectory, create a file with these contents:

```
Man=mwanamume
Woman=mwanamke
```

And save the file with a .txt extension (say, as "Swahili.txt"). Use the command-line resource generator tool to turn this file into a binary resource file that follows the naming convention ***MainAssembly.languagetag*.resources**. For this example, the command is:

```
resgen swahili.txt International.sw.resources
```

The **.resources** file now has to be converted into a satellite assembly named ***MainAssembly*.resources.dll**. This is done with the *assembly linker* tool **al**. Both of these lines should be typed as a single command:

```
al /t:lib /embed:International.sw.resources
/culture:sw /out:International.resources.dll
```

The resulting .DLL should still be in the \sw subdirectory. Do the same process in the \en-US directory after creating an appropriate text file:

```
resgen american.txt International.en-US.resources

al /t:lib /embed:International.en-US.resources /culture:en-
US /out:International.resources.dll
```

Switch back to the parent directory and run **International**. Now, when you run the program, the **man** and **woman** labels should switch between Swahili and American in response to the radio buttons.

Constant resources

While culturally appropriate resources use satellite assemblies, it may be the case that you wish to have certain resources such as graphics and icons embedded directly in the main assembly. Using graphics as resources is a little more difficult than using text because you must use a utility class to generate the resource file. Here's an example command-line class that takes two command-line arguments: the name of a graphics file and the name of the desired resources file:

```
//:c14:GrafResGen.cs
//Generates .resource file from a graphics file
//Usage: GrafResGen [inputFile] [outputFile]
using System.IO;
using System.Resources;
using System.Drawing;

class GrafResGen {
  GrafResGen(
    string name, Stream inStr, Stream outStr){
    ResourceWriter rw = new ResourceWriter(outStr);

    Image img = new Bitmap(inStr);
```

```
    rw.AddResource(name, img);
    rw.Generate();
  }

  public static void Main(string[] args){
    FileStream inF = null;
    FileStream outF = null;
    try {
      string name = args[0];
      inF = new FileStream(name, FileMode.Open);
      string outName = args[1];
      outF =
        new FileStream(outName, FileMode.Create);
      GrafResGen g =
        new GrafResGen(name, inF, outF);
    } finally {
      inF.Close();
      outF.Close();
    }
  }
}///:~
```

A **ResourceWriter** generates binary **.resource** files to a given **Stream**. A **ResXResourceWriter** (not demonstrated) can be used to create an XML representation of the resources that can then be compiled into a binary file using the **resgen** process described above (an XML representation is not very helpful for binary data, so we chose to use a **ResourceWriter** directly).

To use this program, copy an image to the local directory and run:

```
GrafResGen someimage.jpg ConstantResources.resources
```

This will generate a binary resources file that we'll embed in this example program:

```
//:c14:ConstantResources.cs
/*
Compile with: csc /res:ConstantResources.resources
  ConstantResources.cs
*/
//Loads resources from the current assembly
using System.Resources;
using System.Drawing;
```

```
using System.Windows.Forms;

class ConstantResources:Form {
  ConstantResources(){
    PictureBox pb = new PictureBox();
    pb.Dock = DockStyle.Fill;
    Controls.Add(pb);

    ResourceManager rm =
      new ResourceManager(this.GetType());
    pb.Image =
      (Image) rm.GetObject("someimage.jpg");
  }

  public static void Main(){
    Application.Run(new ConstantResources());
  }
}///:~
```

The code in **ConstantResources** is very similar to the code used to load cultural resources from satellites, but without the **GetResourceSet( )** call to load a particular satellite. Instead, the **ResourceManager** looks for resources associated with the **ConstantResources** type. Naturally, those are stored in the **ConstantResources.resources** file generated by the **GrafResGen** utility just described. For the **ResourceManager** to find this file, though, the resource file must be linked into the main assembly in this manner:

```
csc /res:ConstantResources.res ConstantResources.cs
```

Assuming that the resources have been properly embedded into the **ConstantResources.exe** assembly, the **ResourceManager** can load the "**someimage.jpg**" resource and display it in the **PictureBox pb**.

What about the XP look?

If you have been running the sample programs under Windows XP, you may have been disappointed to see that **Control**s do not automatically support XP's graphical themes. In order to activate XP-themed controls, you must set your **Control**'s **FlatStyle** property to **FlatStyle.System** and specify that your program requires Microsoft's **comctl6** assembly to run. You do that by creating another type of non-code resource for your file: a manifest. A manifest is an XML document that specifies all sorts of meta-information about your program: it's name, version, and so forth. One thing you can specify in a manifest is a

dependency on another assembly, such as **comctl6**. To link to **comctl6**, you'll need a manifest of this form:

```
<?xml
   version="1.0" encoding="UTF-8" standalone="yes"
?>
<!-- XPThemed.exe.manifest -->
<assembly
  xmlns="urn:schemas-microsoft-com:asm.v1"
  manifestVersion="1.0">
   <assemblyIdentity
      type="win32"
      name="Thinkingin.Csharp.C13.XPThemes"
      version="1.0.0.0"
      processorArchitecture = "X86"
   />
   <description>Demonstrate XP Themes</description>
   <!-- Link to comctl6 -->
   <dependency>
      <dependentAssembly>
         <assemblyIdentity
           type="win32"
           name="Microsoft.Windows.Common-Controls"
           version="6.0.0.0"
           processorArchitecture="X86"
           publicKeyToken="6595b64144ccf1df"
           language="*"
         />
     </dependentAssembly>
  </dependency>
</assembly>
```

The manifest file is an XML-formatted source of meta-information about your program. In this case, after specifying our own **assemblyIdentity,** we specify the dependency on **Common-Controls** version 6.

Name this file *programName***.exe.manifest** and place it in the same directory as your program. If you do, the .NET Runtime will automatically give the appropriate **Control**s in your program XP themes. Here's an example program:

```
//:c14:XPThemed.cs
using System.Windows.Forms;
using System.Drawing;
```

```
class XPThemed: Form {
  XPThemed(){
    ClientSize = new Size(250, 100);
    Button b = new Button();
    b.Text = "XP Style";
    b.Location = new Point(10, 10);
    b.FlatStyle = FlatStyle.System;
    Controls.Add(b);

    Button b2 = new Button();
    b2.Text = "Standard";
    b2.Location = new Point(100, 10);
    Controls.Add(b2);
  }

  public static void Main(){
    Application.Run(new XPThemed());
  }
}///:~
```

When run without an appropriate manifest file, both buttons will have a default gray style:

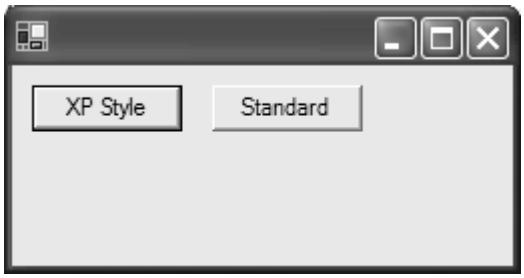


Figure 14-7: By default, Windows Forms do not use XP styles

When **XPThemed.exe.manifest** is available, **b** will use the current XP theme, while **b2,** whose **FlatStyle** is the default **FlatStyle.Standard**, will not.



Figure 14-8: A manifest file provides access to the XP look-and-feel

Fancy buttons

In addition to creating theme-aware buttons, it is an easy matter to create buttons that have a variety of graphical features and that change their appearance in response to events. In order to run this example program, you'll have to have four images in the active directory (in the example code, they're assumed to be named "tic.gif", "away.gif","in.gif", and "hover.gif").

```
//:c14:ButtonForm.cs
///Demonstrates various types of buttons
using System.Windows.Forms;
using System;
using System.Collections;
using System.Drawing;

class ButtonForm : Form {

  ButtonForm() {
    ClientSize = new System.Drawing.Size(400, 200);
    Text = "Buttons, in all their glory";

    Button simple = new Button();
    simple.Text = "Simple";
    simple.Location = new Point(10, 10);

    Button image = new Button();
    image.Image = Image.FromFile(".\\TiC.gif");
    image.Text = "Text";
    image.Location = new Point(120, 10);

    Button popup = new Button();
```

```
    popup.Location = new Point(230, 10);
    popup.Text = "Popup";
    popup.FlatStyle = FlatStyle.Popup;

    FlyOverButton flyOver =
      new FlyOverButton("Away", "In", "Hovering");
    flyOver.Location = new Point(10, 40);

    FlyOverImages flyOverImages =
      new FlyOverImages(
        ".\\away.gif",".\\in.gif",".\\hover.gif");
    flyOverImages.Location = new Point(230, 40);

    Controls.AddRange(new Control[]{
      simple, image, popup, flyOver, flyOverImages});
  }

  public static void Main() {
    Application.Run(new ButtonForm());
  }
}

class FlyOverButton : Button {
  string away;
  string inStr;
  string hover;
  internal FlyOverButton(
    string away, string inStr, string hover) {
    this.away = away;
    this.inStr = inStr;
    this.hover = hover;
    FlatStyle = FlatStyle.Popup;
    Text = away;
    MouseEnter += new EventHandler(OnMouseEnter);
    MouseHover += new EventHandler(OnMouseHover);
    MouseLeave += new EventHandler(OnMouseLeave);
  }
  private void OnMouseEnter(
    object sender, System.EventArgs args) {
    ((Control)sender).Text = inStr;
  }
```

```
  private void OnMouseHover(
    object sender, System.EventArgs args) {
    ((Control)sender).Text = hover;
  }

  private void OnMouseLeave(
    object sender, System.EventArgs args) {
    ((Control)sender).Text = away;
  }
}

class FlyOverImages : Button {
  internal FlyOverImages(
    string away, string inStr, string hover) {
    ImageList = new ImageList();
    ImageList.Images.Add(Image.FromFile(away));
    ImageList.Images.Add(Image.FromFile(inStr));
    ImageList.Images.Add(Image.FromFile(hover));
    FlatStyle = FlatStyle.Popup;
    ImageIndex = 0;
    MouseEnter += new EventHandler(OnMouseEnter);
    MouseHover += new EventHandler(OnMouseHover);
    MouseLeave += new EventHandler(OnMouseLeave);
  }
  private void OnMouseEnter(
    object sender, System.EventArgs args) {
    ((Button)sender).ImageIndex = 1;
  }

  private void OnMouseHover(
    object sender, System.EventArgs args) {
    ((Button)sender).ImageIndex = 2;
  }

  private void OnMouseLeave(
    object sender, System.EventArgs args) {
    ((Button)sender).ImageIndex = 0;
  }
}///:~
```

The first button created and placed on the form is **simple** and its appearance and behavior should be familiar. The second button **image**, sets its **Image** property from an **Image** loaded from a file. The same **Image** is displayed at all times; if the **Text** property is set, the label will be drawn *over* the **Image**.

The third button **popup** has a **FlatStyle** of **FlatStyle.Popup**. This button appears flat until the mouse passes over it, at which point it is redrawn with a 3-D look.

The fourth and fifth buttons require more code and so are written as their own classes: **FlyOverButton** and **FlyOverImages**. The **FlyOverButton** is a regular button, but has event handlers for **MouseEnter**, **MouseHover**, and **MouseLeave** which set the **Text** property as appropriate.

FlyOverImages takes advantage of the **ImageList** property of **Button**. Like **FlyOverButton**, **FlyOverImages** uses mouse events to change the image displayed on the button, but instead of manipulating the **Image** property, it sets the **ImageIndex** property, which correspond indices in the **ImageList** configured in the **FlyOverImages( )** constructor.

Tooltips

The one "fancy" thing that the previous example did not show is probably the one you most expect – the help text that appears when the mouse hovers over a control for more than a few moments. Such tooltips are, surprisingly, not a property of the **Control** above which they appear, but rather are controlled by a separate **ToolTip** object. This would seem to violate a design rule-of-thumb: Objects should generally contain a navigable reference to all objects externally considered associated. As a user or programmer, one would definitely consider the tooltip to be "part of" what distinguishes one control from another, so one should expect a **Tooltip** property in **Control**. Another surprise is that the **ToolTip** does not conform to the containment model of Windows Forms, it is not placed within the **Controls** collection of another **Control**. This is an example of how even the best-designed libraries (and Windows Forms is top-notch) contain inconsistencies and quirks; while it can be very helpful to study the design of a good library to aid your design education, all libraries contain questionable choices.

Adding a **ToolTip** to a **Control** requires that a reference to the **Control** and its desired text be passed to an instance of **ToolTip** (which presumably maintains an internal **IDictionary,** which begs the question of why a **ToolTip** instance is required rather than using a static method). Here's an example that shows the basic use of a **ToolTip**:

```
//:c14:TooltipDisplay.cs
using System;
using System.Drawing;
using System.Windows.Forms;

class TooltipDisplay : Form {
  TooltipDisplay(){
    Button b = new Button();
    b.Text = "Button";
    b.Location = new Point(10, 10);
    Controls.Add(b);

    ToolTip t = new ToolTip();
    t.SetToolTip(b, "Does nothing");
  }

  public static void Main(){
    Application.Run(new TooltipDisplay());
  }
}///:~
```

Displaying and editing text

One of the most common tasks for a GUI is displaying formatted text. In Windows Forms, formatted text is the realm of the **RichTextBox**, which displays and manipulates text in Rich Text Format. The details of the RTF syntax are thankfully hidden from the programmer, text appearance is manipulated using various **Selection*xxx*** properties, which manipulate the chosen substring of the total **RichTextBox.Text**. You can even, if you're so inclined, get and set the full RTF text (which is actually helpful when dragging-and-dropping from, say, Word. Drag-and-drop is covered later in this chapter.)

This example allows you to add arbitrary text to a **RichTextBox** with various formatting options chosen semirandomly:

```
//:c14:TextEditing.cs
///Demonstrates the TextBox and RichTextBox controls
using System.Windows.Forms;
using System;
using System.Collections;
using System.Drawing;
```

```
class TextEditing : Form {
  TextBox tb;
  RichTextBox rtb;
  Random rand = new Random();

  TextEditing() {
    ClientSize = new Size(450, 400);
    Text = "Text Editing";

    tb = new TextBox();
    tb.Text = "Some Text";
    tb.Location = new Point(10, 10);

    Button bold = new Button();
    bold.Text = "Bold";
    bold.Location = new Point(350, 10);
    bold.Click += new EventHandler(bold_Click);

    Button color = new Button();
    color.Text = "Color";
    color.Location = new Point(350, 60);
    color.Click += new EventHandler(color_Click);

    Button size = new Button();
    size.Text = "Size";
    size.Location = new Point(350, 110);
    size.Click += new EventHandler(size_Click);

    Button font = new Button();
    font.Text = "Font";
    font.Location =new Point(350, 160);
    font.Click += new EventHandler(font_Click);

    rtb = new RichTextBox();
    rtb.Location = new Point(10, 50);
    rtb.Size = new Size(300, 180);

    Controls.AddRange(
      new System.Windows.Forms.Control[]{
        tb, rtb, bold, color, size, font});
  }
```

```
private void AddAndSelectText() {
  string newText = tb.Text + "\n";
  int insertionPoint = rtb.SelectionStart;
  rtb.AppendText(newText);
  rtb.SelectionStart = insertionPoint;
  rtb.SelectionLength = newText.Length;
}

private void ResetSelectionAndFont() {
  /* Setting beyond end of textbox places
   insertion at end of text */
  rtb.SelectionStart = Int16.MaxValue;
  rtb.SelectionLength = 0;
  rtb.SelectionFont =
  new Font("Verdana", 10, FontStyle.Regular);
  rtb.SelectionColor = Color.Black;
}
private void bold_Click(
  object sender, System.EventArgs e) {
  AddAndSelectText();
  rtb.SelectionFont =
    new Font("Verdana", 10, FontStyle.Bold);
  ResetSelectionAndFont();
}

private void color_Click(
  object sender, System.EventArgs e) {
  AddAndSelectText();
  rtb.SelectionColor =
    (rand.NextDouble()) > 0.5 ?
  Color.Red : Color.Blue;
  ResetSelectionAndFont();
}

private void size_Click(
  object sender, System.EventArgs e) {
  AddAndSelectText();
  int fontSize = 8 + rand.Next(10);
  rtb.SelectionFont =
```

```
      new Font("Verdana", fontSize,
        FontStyle.Regular);
    ResetSelectionAndFont();
  }

  private void font_Click(
    object sender, System.EventArgs e) {
    AddAndSelectText();
    FontFamily[] families = FontFamily.Families;
    int iFamily = rand.Next(families.Length);
    rtb.SelectionFont =
      new Font(families[iFamily], 10,
        FontStyle.Regular);
    ResetSelectionAndFont();
  }

  static void Main() {
    Application.Run(new TextEditing());
  }
}///:~
```

Everything in the **TextEditing( )** constructor should be familiar: A number of **Button**s are created, event handlers attached, and a **TextBox** and **RichTextBox** are placed on the **Form** as well.

The methods **AddAndSelectText( )** and **ResetSelectionAndFont( )** are used by the various event handlers. In **AddAndSelectText( )** the text to be inserted is taken from the **TextBox tb** and a newline added. The current **rtb.SelectionStart** is remembered, the new text appended to the **RichTextBox,** and the selection is set to begin with the remembered **insertionPoint** and **SelectionLength** to the length of the inserted text.

ResetSelectionAndFont( ) sets the insertion point at the end of the text by giving it an impossibly high value. The selection is reset to use the default font (10 pt. Verdana in black) using the appropriate properties.

The various event handlers call **AddAndSelectText( )** and then manipulate various aspects of the selected text – different sizes, colors, and fonts are randomly chosen.

Linking text

In the past decade, hypertext has gone from an esoteric topic to probably the dominant form of human-computer interaction. However, incorporating text links into a UI has been a big challenge. Windows Forms changes that with its **LinkLabel** control. The **LinkLabel** has powerful support for linking, allowing any number of links within the label area. The **LinkLabel** facilitates the creation of even complex linking semantics, such as the XLink standard (*http://www.w3.org/TR/xlink/*).

While it's possible to use a **LinkLabel** to activate other Windows Forms behavior, the most common use is likely to be activating the full-featured Web browser. To do that, we need to introduce the **Process** class from the **System.Diagnostics** namespace. The **Process** class provides thorough access to local and remote processes, but the core functionality is starting a local process, i.e., launching another application while your application continues to run (or shuts down – the launched process is independent of your application). There are three overloaded versions of **Process.Start( )** that provide various degrees of control over the launched application. The most basic **Process.Start( )** method just takes a **string** and uses the OS's underlying mechanism to determine the appropriate way to run the request; if the string specifies a non-executable file, the extension may be associated with a program and, if so, that program will open it. For instance, **ProcessStart("Foo.cs")** will open the editor associated with the **.cs** extension.

The most advanced **Process.Start( )** takes a **ProcessStartInfo** info. **ProcessStartInfo** contains properties for setting environment variables, whether a window should be shown and in what style, input and output redirection, etc. This example uses the third overload of **Process.Start( )**, which takes an application name and a string representing the command-line arguments, to launch Internet Explorer and surf to www.ThinkingIn.Net.

```
//:c14:LinkLabelDemo.cs
//Demonstrates the LinkLabel
using System;
using System.Drawing;
using System.Diagnostics;
using System.Windows.Forms;

class LinkLabelDemo : Form {
  LinkLabelDemo(){
    LinkLabel label1 = new LinkLabel();
```

```
    label1.Text = "Download Thinking in C#";
    label1.Links.Add(9, 14,
      "http://www.ThinkingIn.Net/");
    label1.LinkClicked +=
    new LinkLabelLinkClickedEventHandler(
      InternetExplorerLaunch);
    label1.Location = new Point(10, 10);
    label1.Size = new Size(160, 30);
    Controls.Add(label1);

    LinkLabel label2 = new LinkLabel();
    label2.Text = "Show message";
    label2.Links.Add(0, 4, "Foo");
    label2.Links.Add(5, 8, "Bar");
    label2.LinkClicked +=
      new LinkLabelLinkClickedEventHandler(
        MessageBoxShow);
    label2.Location = new Point(10, 60);
    Controls.Add(label2);
  }

  public void InternetExplorerLaunch(
    object src, LinkLabelLinkClickedEventArgs e){
    string url = (string) e.Link.LinkData;
    Process.Start("IExplore.exe", url);
    e.Link.Visited = true;
  }

  public void MessageBoxShow(
    object src, LinkLabelLinkClickedEventArgs e){
    string msg = (string) e.Link.LinkData;
    MessageBox.Show(msg);
    e.Link.Visited = true;
  }

  public static void Main(){
    Application.Run(new LinkLabelDemo());
  }
}///:~
```

Not all the **LinkLabel.Text** need be a link; individual links are added to the **Links** collection by specifying an offset and the length of the link. If you do not

need any information other than the fact that the link was clicked, you do not need to include the third argument to **Links.Add( )**, but typically you will store some data to be used by the event handler. In the example, the phrase "Thinking in C#" is presented underlined, in the color of the **LinkColor** property (by default, this is set by the system and is usually blue). This link has as its associated data, a URL. **Label2** has two links within it, one associated with the **string** "foo" and the other with "bar."

While a **LinkLabel** can have many links, all the links share the same event handler (of course, it's a multicast delegate, so you can add as many methods to the event-handling chain as desired, but you cannot directly associated a specific event-handling method with a specific link). The **Link** is passed to the event handler via the event arguments, so the delegate is the descriptively named **LinkLabelLinkClickedEventHandler**. The **LinkData** (if it was specified in the **Link**'s constructor) can be any **object**. In the example, we downcast the **LinkData** to **string**.

The **InternetExplorerLaunch( )** method uses **Process.Start( )** to launch Microsoft's Web browser. **MessageBoxShow( )** demonstrates the convenient **MessageBox** class, which pops up a simple alert dialog. At the end of the event handlers, the appropriate **Link** is set to **Visited**, which redraws the link in the **LinkLabel**.**VisitedLinkColor**.

Checkboxes **and** RadioButtons

As briefly mentioned in the **International** example, radio buttons in Windows Forms have the simple model of being mutually exclusive within their containing **Controls** collection. Sometimes it is sufficient to just plunk some radio buttons down on a **Form** and be done with it, but usually you will use a **Panel** or **GroupBox** to contain a set of logically related radio buttons (or other controls). Usually, a set of related **RadioButton**s should have the same event handler since generally the program needs to know "Which of the radio buttons in the group is selected?" Since **EventHandler** delegates pass the source **object** as the first parameter in their arguments, it is easy for a group of buttons to share a single delegate method and use the **src** argument to determine which button has been activated. The use of a **GroupBox** and this form of sharing a delegate method is demonstrated in the next example.

CheckBox controls are not mutually exclusive; any number can be in any state within a container. **CheckBox**es cycle between two states (**CheckState.Checked** and **CheckState.Unchecked**) by default, but by

setting the **ThreeState** property to true, can cycle between **Checked**, **Unchecked**, and **CheckState.Indeterminate**.

This example demonstrates a standard **CheckBox**, one that uses an **Image** instead of text (like **Button**s and many other **Control**s, the default appearance can be changed using a variety of properties), a three-state **CheckBox**, and grouped, delegate-sharing **RadioButton**s:

```
//:c14:CheckAndRadio.cs
//Demonstrates various types of buttons
using System.Windows.Forms;
using System;
using System.Collections;
using System.Drawing;

class CheckAndRadio : Form {
  CheckAndRadio() {
    ClientSize = new System.Drawing.Size(400, 200);
    Text = "Checkboxes and Radio Buttons";

    CheckBox simple = new CheckBox();
    simple.Text = "Simple";
    simple.Location = new Point(10, 10);
    simple.Click +=
      new EventHandler(OnSimpleCheckBoxClick);

    CheckBox image = new CheckBox();
    image.Image = Image.FromFile(".\\TiC.gif");
    image.Location = new Point(120, 10);
    image.Click +=
      new EventHandler(OnSimpleCheckBoxClick);

    CheckBox threeState = new CheckBox();
    threeState.Text = "Three state";
    threeState.ThreeState = true;
    threeState.Location = new Point(230, 10);
    threeState.Click +=
      new EventHandler(OnThreeStateCheckBoxClick);

    Panel rbPanel = new Panel();
    rbPanel.Location = new Point(10, 50);
    rbPanel.Size = new Size(420, 50);
```

```
    rbPanel.AutoScroll = true;

    RadioButton f1 = new RadioButton();
    f1.Text = "Vanilla";
    f1.Location = new Point(0, 10);
    f1.CheckedChanged +=
      new EventHandler(OnRadioButtonChange);
    RadioButton f2 = new RadioButton();
    f2.Text = "Chocolate";
    f2.Location = new Point(140, 10);
    f2.CheckedChanged +=
      new EventHandler(OnRadioButtonChange);
    RadioButton f3 = new RadioButton();
    f3.Text = "Chunky Monkey";
    f3.Location = new Point (280, 10);
    f3.CheckedChanged +=
      new EventHandler(OnRadioButtonChange);
    f3.Checked = true;

    rbPanel.Controls.AddRange(
      new Control[]{ f1, f2, f3});

    Controls.AddRange(
      new Control[]{
        simple, image, threeState, rbPanel});
  }

  private void OnSimpleCheckBoxClick(
    object sender, EventArgs args){
    CheckBox cb = (CheckBox) sender;
    Console.WriteLine(
      cb.Text + " is " + cb.Checked);
  }

  private void OnThreeStateCheckBoxClick(
      object sender, EventArgs args){
    CheckBox cb = (CheckBox) sender;
    Console.WriteLine(
      cb.Text + " is " + cb.CheckState);
  }
```

```
  private void OnRadioButtonChange(
    object sender, EventArgs args){
    RadioButton rb = (RadioButton) sender;
    if (rb.Checked == true)
      Console.WriteLine(
        "Flavor is " + rb.Text);
  }

  public static void Main() {
    Application.Run(new CheckAndRadio());
  }
}///:~
```

List, Combo, **and** CheckedListBoxes

Radio buttons and check boxes are appropriate for selecting among a small number of choices, but the task of selecting from larger sets of options is the work of the **ListBox**, the **ComboBox**, and the **CheckedListBox**.

This example shows the basic use of a **ListBox**. This **ListBox** allows for only a single selection to be chosen at a time; if the **SelectionMode** property is set to **MultiSimple** or **MultiExtended**, multiple items can be chosen (**MultiExtended** should be used to allow SHIFT, CTRL, and arrow shortcuts). If the selection mode is **SelectionMode.Single**, the **Item** property contains the one-and-only selected item, for other modes the **Items** property is used.

```
//:c14:ListBoxDemo.cs
//Demonstrates ListBox selection
using System;
using System.Drawing;
using System.Windows.Forms;

class ListBoxDemo : Form {
  ListBoxDemo(){
    ListBox lb = new ListBox();
    for (int i = 0; i < 10; i++) {
      lb.Items.Add(i.ToString());
    }
    lb.Location = new Point(10, 10);
    lb.SelectedValueChanged +=
      new EventHandler(OnSelect);
    Controls.Add(lb);
```

```
  }

  public void OnSelect(object src, EventArgs ea){
    ListBox lb = (ListBox) src;
    Console.WriteLine(lb.SelectedItem);
  }

  public static void Main(){
    Application.Run(new ListBoxDemo());
  }
}///:~
```

The **ComboBox** is similarly easy to use, although it can only be used for single selection. This example demonstrates the **ComboBox**, including its ability to sort its own contents:

```
//:c14:ComboBoxDemo.cs
///Demonstrates the ComboBox
using System;
using System.Drawing;
using System.Windows.Forms;

class ComboBoxDemo : Form {
  ComboBox presidents;
  CheckBox sorted;

  ComboBoxDemo() {
    ClientSize = new Size(320, 200);
    Text = "ComboBox Demo";

    presidents = new ComboBox();
    presidents.Location = new Point(10, 10);
    presidents.Items.AddRange(
      new string[]{
        "Washington", "Adams J", "Jefferson",
        "Madison", "Monroe", "Adams JQ", "Jackson",
        "Van Buren", "Harrison", "Tyler", "Polk",
        "Taylor", "Fillmore", "Pierce", "Buchanan",
        "Lincoln", "Johnson A", "Grant", "Hayes",
        "Garfield", "Arthur", "Cleveland",
        "Harrison", "McKinley", "Roosevelt T",
        "Taft", "Wilson", "Harding", "Coolidge",
```

```
        "Hoover", "Roosevelt FD", "Truman",
        "Eisenhower", "Kennedy", "Johnson LB",
        "Nixon", "Ford", "Carter", "Reagan",
        "Bush G", "Clinton", "Bush GW"});
    presidents.SelectedIndexChanged +=
      new EventHandler(OnPresidentSelected);

    sorted = new CheckBox();
    sorted.Text = "Alphabetically sorted";
    sorted.Checked = false;
    sorted.Click +=
      new EventHandler(NonReversibleSort);
    sorted.Location = new Point(150, 10);

    Button btn = new Button();
    btn.Text = "Read selected";
    btn.Click += new EventHandler(GetPresident);
    btn.Location = new Point(150, 50);

    Controls.AddRange(
      new Control[]{presidents, sorted, btn});
  }

  private void NonReversibleSort(
    object sender, EventArgs args) {
    //bug, since non-reversible
    presidents.Sorted = sorted.Checked;
  }

  private void OnPresidentSelected(
    object sender, EventArgs args) {
    int selIdx = presidents.SelectedIndex;
    if (selIdx > -1) {
      Console.WriteLine(
        "Selected president is: "
        + presidents.Items[selIdx]);
    } else {
      Console.WriteLine(
        "No president is selected");
    }
  }
```

```
  private void GetPresident(
    object sender, EventArgs args) {
    //Doesn't work, since can be blank
    // or garbage value
    Console.WriteLine(presidents.Text);
    //So you have to do something like this...
    string suggestion = presidents.Text;
    if (presidents.Items.Contains(suggestion)) {
      Console.WriteLine(
        "Selected president is: " + suggestion);
    } else {
      Console.WriteLine(
        "No president is selected");
    }
  }

  public static void Main() {
    Application.Run(new ComboBoxDemo());
  }
}///:~
```

After the names of the presidents are loaded into the **ComboBox**, a few handlers are defined: the checkbox will trigger **NonReversibleSort( )** and the button will trigger **GetPresident( )**. The implementation of **NonReversibleSort( )** sets the **ComboBox**'s **Sorted** property depending on the selection state of the **sorted Checkbox**. This is a defect as, once sorted, setting the **Sorted** property to **false** will *not* return the **ComboBox** to its original chronologically-ordered state.

GetPresident( ) reveals another quirk. The value of **ComboBox.Text** is the value of the editable field in the **ComboBox**, even if no value has been chosen, or if the user has typed in non-valid data. In order to confirm that the data in **ComboBox.Text** is valid, you have to search the **Items** collection for the text, as demonstrated.

The **CheckedListBox** is the most complex of the list-selection controls. This example lets you specify your musical tastes, printing your likes and dislikes to the console.

```
//:c14:CheckedListBoxDemo.cs
///Demonstrates the CheckedListBox
using System;
```

```
using System.Drawing;
using System.Windows.Forms;

class CheckedListBoxDemo : Form {
  CheckedListBox musicalTastes;

  CheckedListBoxDemo(){
    ClientSize = new Size(320, 200);
    Text = "CheckedListBox Demo";

    musicalTastes = new CheckedListBox();
    musicalTastes.Location = new Point(10, 10);
    musicalTastes.Items.Add(
      "Classical", CheckState.Indeterminate);
    musicalTastes.Items.Add(
      "Jazz", CheckState.Indeterminate);
    musicalTastes.Items.AddRange(
      new string[]{
        "Blues", "Rock", "Punk", "Grunge",
        "Hip hop"});

    MakeAllIndeterminate();

    Button getTastes = new Button();
    getTastes.Location = new Point(200, 10);
    getTastes.Width = 100;
    getTastes.Text = "Get tastes";
    getTastes.Click +=
      new EventHandler(OnGetTastes);

    Controls.Add(musicalTastes);
    Controls.Add(getTastes);
  }

  private void MakeAllIndeterminate(){
    CheckedListBox.ObjectCollection items =
      musicalTastes.Items;
    for (int i = 0; i < items.Count; i++) {
      musicalTastes.SetItemCheckState(
        i, CheckState.Indeterminate);
    }
```

```
  }

  private void OnGetTastes(object o, EventArgs args){
    //Returns checked _AND_ indeterminate!
    CheckedListBox.CheckedIndexCollection
      checkedIndices = musicalTastes.CheckedIndices;
    foreach(int i in checkedIndices){
      if (musicalTastes.GetItemCheckState(i)
          != CheckState.Indeterminate) {
        Console.WriteLine(
          "Likes: " + musicalTastes.Items[i]);
      }
    }

    //Or, to iterate over the whole collection
    for (int i = 0;
        i < musicalTastes.Items.Count; i++) {
      if (musicalTastes.GetItemCheckState(i)
          == CheckState.Unchecked) {
        Console.WriteLine(
          "Dislike: " + musicalTastes.Items[i]);
      }
    }
  }
  public static void Main(){
    Application.Run(new CheckedListBoxDemo());
  }
}///:~
```

The **CheckedListBoxDemo( )** constructor shows that items can be added either one at a time, with their **CheckState** property defined, or en masse.

The **OnGetTastes( )** method shows a defect in the **CheckedListBox** control; the **CheckedIndices** property returns not just those items with **CheckState.Checked**, but also those with **CheckState.Indeterminate**! An explicit check must be added to make sure that the value at the index really *is* checked. Once an appropriate index is in hand, the value can be retrieved by using the **Items[ ]** array operator.

Multiplane displays with the Splitter **control**

It is often desirable to allow the user to change the proportion of a window devoted to various logical groups; windows with such splits are often said to be divided into multiple "panes." The **Splitter** control allows the user to resize controls manually.

We discussed the use of **GroupBox** as one way to logically group controls such as **RadioButton**s. A more general-purpose grouping control is the **Panel**, which by default is invisible. By placing logically related **Control**s on a **Panel**, and then associating a **Splitter** with the **Panel**, multipane UIs can be created easily. A **Splitter** is associated with a **Control** via the **Dock** property – the **Splitter** resizes the **Control** placed immediately *after* it in the container. The **Control** to be resized and the **Splitter** should be assigned the same **Dock** value.

In this example, we use **Panel**s that we make visible by setting their **BackColor** properties.

```
//:c14:SplitterDemo.cs
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitterDemo : Form {
  SplitterDemo(){
    Panel r = new Panel();
    r.BackColor = Color.Red;
    r.Dock = DockStyle.Left;
    r.Width = 200;

    Panel g = new Panel();
    g.BackColor = Color.Green;
    g.Dock = DockStyle.Fill;

    Panel b = new Panel();
    b.BackColor = Color.Blue;
    b.Dock = DockStyle.Right;
    b.Width = 200;

    Splitter rg = new Splitter();
```

```
    //Set dock to same as resized control (p1)
    rg.Dock = DockStyle.Left;

    Splitter gb = new Splitter();
    //Set dock to same as resized control (p3)
    gb.Dock = DockStyle.Right;

    Controls.Add(g);

    //Splitter added _before_ panel
    Controls.Add(gb);
    Controls.Add(b);

    //Splitter added _before_ panel
    Controls.Add(rg);
    Controls.Add(r);

    Width = 640;
  }

  public static void Main(){
    Application.Run(new SplitterDemo());
  }
}///:~
```

After creating panels **r**, **g**, and **b** and setting their **BackColor**s appropriately, we create a **Splitter rg** with the same **Dock** value as the **r Panel** and another called **gb** with the same **Dock** value as **b**. It is critical that the **Splitter**s are added to the **Form** immediately prior to the **Panel**s they resize. The example starts with **r** and **b** at their preferred **Width** of 200 pixels, while the entire **Form** is set to take 640. However, you can resize the **Panel**s manually.

TreeView **and** ListView

Everyone seems to have a different idea of what the ideal **TreeView** control should look like and every discussion group for every UI toolkit is regularly swamped with the intricacies of **TreeView** programming. Windows Forms is no exception, but the general ease of programming Windows Forms makes basic **TreeView** programming fairly straightforward. The core concept of the **TreeView** is that a **TreeView** contains **TreeNodes**, which in turn contain other **TreeNodes**. This model is essentially the same as the Windows Forms model in which **Control**s contain other **Control**s. So just as you start with a

Form and add **Control**s and **Control**s to those **Control**s, so too you create a **TreeView** and add **TreeNode**s and **TreeNode**s to those **TreeNode**s.

However, the general programming model for Windows Forms is that events are associated with the pieces that make up the whole (the **Control**s within their containers), while the programming model for the **TreeView** is that events are associated with the whole; **TreeNode**s have no events.

This example shows the simplest possible use of **TreeView**.

```
//:c14:TreeViewDemo1.cs
//Demonstrates TreeView control
using System;
using System.Drawing;
using System.Windows.Forms;

class TreeViewDemo1 : Form {
  TreeViewDemo1(){
    TreeView tv = new TreeView();
    TreeNode root = new TreeNode("Fish");
    TreeNode cart = new TreeNode("Sharks & Rays");
    TreeNode bony = new TreeNode("Bony fishes");
    tv.Nodes.Add(root);
    root.Nodes.Add(cart);
    root.Nodes.Add(bony);

    tv.AfterSelect +=
      new TreeViewEventHandler(AfterSelectHandler);
    Controls.Add(tv);
  }

  public void AfterSelectHandler(
    object src, TreeViewEventArgs a){
    TreeNode sel = ((TreeView) src).SelectedNode;
    Console.WriteLine(sel);
  }
  public static void Main(){
    Application.Run(new TreeViewDemo1());
  }
}///:~
```

In the constructor, after a **TreeView** control is initialized, three **TreeNode**s are created. The **root** node is added to the **Nodes** collection of the **TreeView**. The

branch nodes **cart** and **bony** are added to the **Nodes** collection, not of the **TreeView**, but of the **root** node. A **TreeViewEventHandler** delegate is created to output the value of the selected node to the console. The delegate is added to the **TreeView**'s **AfterSelect** event – the **TreeView** has a dozen unique events relating to node selection, collapsing and expanding, and changes to the item's checkbox (itself an optional property).

ListView

ListView may finally replace **TreeView** as "most discussed UI widget." The **ListView** is an insanely customizable widget that is similar to the right-hand side of Windows Explorer – items placed within a **ListView** can be viewed in a detail view, as a list, and as grids of large or small icons. Like the **TreeView**, the **ListView** uses a basic containment model: a **TreeView** contains a collection of **ListViewItem**s which do not themselves have events. The **ListViewItem**s are added to the **Items** property of the **ListView**. Various properties of the **ListView** and the individual **ListViewItem**s control the various displays.

Icon views

The **ListView** contains two **ImageList**s, which hold icons (or other small graphics) that can be associated with different types of **ListViewItem**s. One **ImageList** should be assigned to the **SmallImageList** and the other to the **LargeImageList** property of the **ListView**. Corresponding images should be added to both **ImageList**'s at the same offsets, since the selection of *either* **ImageList** is determined by the **ListViewItem**'s **ImageIndex** property. In other words, if the **ListViewItem.ImageIndex** is set to 3, if **ListView.View** is set to **ListView.LargeIcon** the 4th image in **LargeImageList** will be displayed for that **ListViewItem**, while if **ListView.View == View.SmallIcon**, the 4th image in **SmallImageList** will be displayed.

Details view

The details view of a **ListView** consists of a series of columns, the first of which displays the **ListViewItem** and its associated icon from the **SmallImageList**. Subsequent columns display text describing various aspects related to the **ListViewItem**. The text displayed by the 2nd and subsequent columns is determined by the collection of **ListViewSubItem**s in the **ListViewItem**'s **SubItems** property. The column header text is set with the **ListView.Columns** property. The programmer is responsible for coordinating the consistency of column header offsets and the indices of **ListViewSubItem**s.

This example demonstrates the **ListView**'s various modes.

```
//:c14:ListViewDemo.cs
//Demonstrates the ListView control
using System;
using System.IO;
using System.Drawing;
using System.Windows.Forms;

class ListViewDemo : Form {
  ListView lv;

  ListViewDemo(){
    //Set up control panel
    Panel p = new Panel();
    RadioButton[] btn = new RadioButton[]{
      new RadioButton(), new RadioButton(),
      new RadioButton(), new RadioButton()
    };
    btn[0].Checked = true;
    btn[0].Text = "Details";
    btn[1].Text = "Large Icons";
    btn[2].Text = "List";
    btn[3].Text = "Small Icons";
    for (int i = 0; i < 4; i++) {
      btn[i].Location =
      new Point(10, 20 * (i + 1));
      btn[i].CheckedChanged +=
        new EventHandler(SelectView);
    }
    p.Controls.AddRange(btn);
    p.Dock = DockStyle.Left;
    p.Width = 100;

    Splitter s = new Splitter();
    s.Dock = DockStyle.Left;

    //ListView initial stuff
    lv = new ListView();
    lv.Dock = DockStyle.Fill;

    lv.Columns.Add(
      "File Name", 150, HorizontalAlignment.Left);
```

```
    lv.Columns.Add(
      "Size", 150, HorizontalAlignment.Left);
    lv.View = View.Details;

    //Load images
    Image smallCS = Image.FromFile("cs_sm.bmp");
    Image smallExe = Image.FromFile("exe_sm.bmp");
    ImageList il = new ImageList();
    il.Images.Add(smallCS);
    il.Images.Add(smallExe);
    lv.SmallImageList = il;

    Image largeCS = Image.FromFile("cs_lrg.bmp");
    Image largeExe = Image.FromFile("exe_lrg.bmp");
    ImageList il2 = new ImageList();
    il2.Images.Add(largeCS);
    il2.Images.Add(largeExe);
    lv.LargeImageList = il2;

    DirectoryInfo dir = new DirectoryInfo(".");
    foreach(FileInfo f in dir.GetFiles("*.*")){
      string fName = f.Name;
      string size = f.Length.ToString();
      string ext = f.Extension;

      ListViewItem item = new ListViewItem(fName);
      if (ext == ".cs") {
        item.ImageIndex = 0;
      } else {
        item.ImageIndex = 1;
      }
      item.SubItems.Add(size);
      lv.Items.Add(item);
    }

    Controls.Add(lv);
    Controls.Add(s);
    Controls.Add(p);
    Width = 640;
  }
```

```
  public void SelectView(Object src, EventArgs ea){
    RadioButton rb = (RadioButton) src;
    string viewDesired = rb.Text;
    switch (viewDesired) {
    case "Details" :
      lv.View = View.Details;
      break;
    case "Large Icons" :
      lv.View = View.LargeIcon;
      break;
    case "List" :
      lv.View = View.List;
      break;
    case "Small Icons" :
      lv.View = View.SmallIcon;
      break;
    }
  }

  public static void Main(){
    Application.Run(new ListViewDemo());
  }
}///:~
```

The **ListViewDemo( )** constructor defines a **Panel** and **RadioButton** array to select the various **ListView.View** values. A **Splitter** is also defined to allow this panel to be resized. The **ListView lv** is created and its **Dock** property set to **DockStyle.Fill**. We specify that when the **ListView** is switched to details view, the two columns will be labeled "File Name" and "Size," each will initially be 150 pixels wide, and each will display its text left-aligned. We then set **lv**'s initial view state to, in fact, be **View.Details**.

The next section of the constructor, labeled "LoadImages," loads small and then larger images, and places the corresponding images (**cs_sm.bmp** and **cs_lrg.bmp** and **exe_sm.bmp** and **exe_lrg.bmp**) in **ImageList**s which are assigned to the **ListView**'s **SmallImageList** and **LargeImageList** properties.

The **ListView** is populated with the files in the current directory. For each file found, we determine the name, file length, and extension and create a **ListViewItem** with it's "main" **Text** set to the file's name. If the extension is ".cs" we set that **ListViewItem**'s **IconIndex** to correspond to the C# images in the **ListView**'s **ImageList**s, otherwise we set the index to 1.

The **string** value representing the file's length is added to the **SubItems** collection of the **ListViewItem**. The **ListViewItem** itself is always the *first* item in the **SubItems** list; in this case that will appear in the "File Name" column of the **TreeView**. The **size** string will appear in the *second* column of the **TreeView**, under the "Size" heading.

The last portion of the constructor adds the **Control**s to the **ListViewDemo** form in the appropriate order (remember, because there's a **Splitter** involved, the **Splitter** must be added to the form immediately before the **Panel** it resizes).

The **SelectView( )** delegate method, called by the **SelectionChanged** event of the **RadioButton**s, sets **ListView.View** to the appropriate value from the **View** enumeration.

Using the clipboard and drag and drop

Another perennially challenging interface issue is using the Clipboard and supporting drag and drop operations.

Clipboard

Objects on the Clipboard are supposed to be able to transform themselves into a variety of formats, depending on what the pasting application wants. For instance, a vector drawing placed on the Clipboard should be able to transform itself into a bitmap for Paint or a Scalable Vector Graphics document for an XML editor. Data transfer in Windows Forms is mediated by classes which implement the **IDataObject** interface. A consuming application or **Control** first gets a reference to the **Clipboard**'s current **IDataObject** by calling the static method **Clipboard.GetDataObject( )**. Once in hand, the **IDataObject** method **GetDataPresent( )** is called with the desired type in a **string** or **Type** argument. If the **IDataObject** can transform itself into the requested type, **GetDataPresent( )** returns **true**, and **GetData( )** can then be used to retrieve the data in the requested form.

In this example, one **Button** places a **string** on the Clipboard, and the other button puts the Clipboard's content (if available as a **string**) into a **Label**. If you copy an image or other media that cannot be transformed into a **string** onto the Clipboard and attempt to paste it with this program, the label will display "Nothing."

```
//:c14:ClipboardDemo.cs
//Cuts to and pastes from the system Clipboard
```

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;

class ClipboardDemo : Form {
  Label l;

  ClipboardDemo(){
    Button b = new Button();
    b.Location = new Point(10, 10);
    b.Text = "Clip";
    b.Click +=
      new EventHandler(AddToClipboard);
    Controls.Add(b);

    Button b2 = new Button();
    b2.Location = new Point(100, 10);
    b2.Text = "Paste";
    b2.Click +=
      new EventHandler(CopyFromClip);
    Controls.Add(b2);

    l = new Label();
    l.Text = "Nothing";
    l.Location = new Point(100, 50);
    l.Size = new Size(200, 20);
    Controls.Add(l);
  }

  public void AddToClipboard(Object s, EventArgs a){
    Clipboard.SetDataObject("Text. On clipboard.");
  }

  public void CopyFromClip(Object s, EventArgs a){
    IDataObject o = Clipboard.GetDataObject();
    if (o.GetDataPresent(typeof(string))) {
      l.Text = o.GetData(typeof(string)).ToString();
    } else {
      l.Text = "Nothing";
    }
```

```
  }

  public static void Main(){
    Application.Run(new ClipboardDemo());
  }
}///:~
```

Nothing unusual is done in the **ClipboardDemo( )** constructor; the only thing somewhat different from other demos is that the **Label l** is made an instance variable so that we can set its contents in the **CopyFromClip( )** method.

The **AddToClipboard( )** method takes an **object**, in this case a **string**. If the Clipboard is given a **string** or **Image**, that value can be pasted into other applications (use **RichTextBox.Rtf** or **SelectedRtf** for pasting into Microsoft Word).

CopyFromClip( ) must be a little cautious, as there is no guarantee that the Clipboard contains data that can be turned into a **string** (every .NET Framework **object** can be turned into a **string**, but the Clipboard may very well contain data that is not a .NET Framework **object**). If the **IDataObject** instance can indeed be expressed as a **string**, we retrieve it, again specifying **string** as the type we're looking for. Even though we've specified the type in the argument, we still need to change the return value of **GetData( )** from **object** to **string**, which we do by calling the **ToString( )** method (we could also have used a **(string)** cast).

Drag and drop

Drag and drop is a highly visual form of data transfer between components and applications. In Windows Forms, drag and drop again uses the **IDataObject** interface, but also involves visual feedback. In this example, we demonstrate both drag and drop and the creation of a custom **IDataObject** that can transform itself into a variety of types.

An exemplar of something that transforms into a variety of types is, of course, SuperGlo, the floorwax that tastes like a dessert topping (or is it the dessert topping that cleans like a floorwax? All we know for sure is that it's delicious *and* practical!)

```
//:c14:SuperGlo.cs
//Compile with:
//csc SuperGloMover.cs DragAndDropDemo.cs SuperGlo.cs
//Domain objects for custom drag-and-drop
using System;
```

```
interface FloorWax{
  void Polish();
}

interface DessertTopping{
  void Eat();
}

class SuperGlo : FloorWax, DessertTopping {
  public void Polish() {
    Console.WriteLine(
      "SuperGlo makes your floor shine!");
  }
  public void Eat() {
    Console.WriteLine(
      "SuperGlo tastes great!");
  }
}///:~ (example continues with SuperGloMover.cs)
```

In this listing, we define the domain elements – a **FloorWax** interface with a **Polish( )** method, a **DessertTopping** interface with an **Eat( )** method, and the **SuperGlo** class, which implements both the interfaces by writing to the console. Now we need to define an **IDataObject** implementation that is versatile enough to demonstrate **SuperGlo**:

```
//(continuation)
//:c14:SuperGloMover.cs
//Custom IDataObject, can expose SuperGlo as:
//FloorWax, DessertTopping, or text
using System;
using System.Windows.Forms;

class SuperGloMover : IDataObject {
  //Concern 1: What formats are supported?
  public string[] GetFormats() {
    return new string[]{
      "FloorWax", "DessertTopping", "SuperGlo"};
  }

  public string[] GetFormats(bool autoConvert) {
    if (autoConvert) {
      return new string[]{
```

```
      "FloorWax", "DessertTopping",
      "SuperGlo",  DataFormats.Text};
  } else {
    return GetFormats();
  }
}

//Concern 2: Setting the data

//Storage
SuperGlo superglo;

public void SetData(object o) {
  if (o is SuperGlo) {
    superglo = (SuperGlo) o;
  }
}

public void SetData(string fmt, object o) {
  if (fmt == "FloorWax" || fmt == "DessertTopping"
    || fmt == "SuperGlo") {
    SetData(o);
  } else {
    if (fmt == DataFormats.Text) {
      superglo = new SuperGlo();
    } else {
      Console.WriteLine(
        "Can't set data to type " + fmt);
    }
  }
}

public void SetData(Type t, object o) {
  SetData(t.Name, o);
}

public void SetData(
  String fmt, bool convert, object o) {
  if (fmt == DataFormats.Text
    && convert == false) {
    Console.WriteLine(
```

```
        "Refusing to change a string " +
        "to a superglo");
    } else {
      SetData(fmt, o);
    }
  }

  //Concern 3: Is there a format client can use?
  public bool GetDataPresent(string fmt) {
    if (fmt == "DessertTopping" || fmt == "FloorWax"
      || fmt == DataFormats.Text
      || fmt == "SuperGlo") {
      return true;
    } else {
      return false;
    }
  }

  public bool GetDataPresent(Type t) {
    return(GetDataPresent(t.Name));
  }

  public bool GetDataPresent(
    String fmt, bool convert) {
    if (fmt == DataFormats.Text
      && convert == false) {
      return false;
    } else {
      return GetDataPresent(fmt);
    }
  }

  //Concern 4: Get the data in requested format
  public object GetData(string fmt) {
    switch (fmt) {
    case "FloorWax" :
      return superglo;
    case "DessertTopping" :
      return superglo;
    case "SuperGlo" :
      return superglo;
```

```
      case "Text" :
        return "SuperGlo -- It's a FloorWax! "
        + "And a dessert topping!";
      default :
        Console.WriteLine(
          "SuperGlo is many things, but not a "
          + fmt);
        return null;
      }
    }

    public object GetData(Type t) {
      string fmt = t.Name;
      return GetData(fmt);
    }

    public object GetData(string fmt, bool convert) {
      if (fmt == DataFormats.Text
        && convert == false) {
        return null;
      } else {
        return GetData(fmt);
      }
    }
}///:~ (example continues with DragAndDropDemo.cs)
```

To implement **IDataObject**, you must address four concerns: What are the formats that are supported, setting the data, is the data in a format the client can use, and getting the data in the specific format the client wants.

The first concern is addressed by the two overloaded **GetFormats( )** methods. Both return **string** arrays which represent the .NET classes into which the stored data can be transformed. In addition to the types which the **SuperGlo** class actually instantiates, we also specify that **SuperGlo** can automatically be converted to-and-from text. The **DataFormats** class contains almost two dozen static properties defining various Clipboard formats that Windows Forms already understands.

Setting the data is done, in the simplest case, by passing in an **object**, which is stored in the instance variable **superglo**. If a person attempts to store a non-**SuperGlo** object, the request will be quietly ignored. The .NET documentation is silent on whether **SetData( )** should throw an exception if called with an

argument of the wrong type; and typically silence means that one should *not* throw an exception. This goes against the grain of good programming practice (usually, the argument to a method should always either affect the return value of the method, affect the state of the object or the state of the argument, or result in an exception).

The second **SetData( )** method takes a **string** representing a data format and an object. If the **string** is any one of the native types of **SuperGlo** ("SuperGlo," "Dessert," or "FloorWax"), the object is passed to the **SetData(object)** method. If the **string** is set to **DataFormats.Text**, a new **SuperGlo** is instantiated and stored. If the **string** is not one of these four values, a diagnostic is printed and the method fails. The third **SetData( )** method takes a **Type** as its first argument and simply passes that argument's **Name** property forward to **SetData(string, object)**.

The fourth-and-final **SetData( )** method takes, in addition to a format **string** and the data **object** itself, a **bool** that may be used to turn off auto-conversion (in this case, the "conversion" from **DataFormat.Text** that just instantiates a new **SuperGlo**).

The **GetDataPresent( )** methods return **true** if the argument is one of the **SuperGlo** types or **DataTypes.Text** (except if the **convert** argument is set to **false**).

The **GetData( )** method returns a reference to the stored **SuperGlo** if the format requested is "SuperGlo," "FloorWax," or "DessertTopping." If the format requested is "Text," the method returns a promotional reference. If anything else, **GetData( )** returns a **null** reference.

Now that we have the domain objects and **SuperGloMover**, we can put it all together visually:

```
//(continuation)
//:c14:DragAndDropDemo.cs
//Demonstrates drag-and-drop with SuperGlo and
//SuperGloMover
using System;
using System.Drawing;
using System.Windows.Forms;

class DragAndDropDemo : Form {
  DragAndDropDemo() {
    ClientSize = new Size(640, 320);
```

```
    Text = "Drag & Drop";

    Button b = new Button();
    b.Text = "Put SuperGlo on clipboard";
    b.Location = new Point(10, 40);
    b.Width = 100;
    b.Click += new EventHandler(OnPutSuperGlo);

    PictureBox pb = new PictureBox();
    pb.Image = Image.FromFile(@".\superglo.jpg");
    pb.SizeMode = PictureBoxSizeMode.AutoSize;
    pb.Location = new Point(220, 100);
    pb.MouseDown +=
      new MouseEventHandler(OnBeginDrag);

    Panel floor = new Panel();
    floor.BorderStyle = BorderStyle.Fixed3D;
    floor.Location = new Point(10, 80);
    Label f = new Label();
    f.Text = "Floor";
    floor.Controls.Add(f);
    floor.AllowDrop = true;
    floor.DragEnter +=
      new DragEventHandler(OnDragEnterFloorWax);
    floor.DragDrop +=
      new DragEventHandler(OnDropFloorWax);

    Panel dessert = new Panel();
    dessert.BorderStyle = BorderStyle.Fixed3D;
    dessert.Location = new Point(300, 80);
    Label d = new Label();
    d.Text = "Dessert";
    dessert.Controls.Add(d);
    dessert.AllowDrop = true;
    dessert.DragEnter +=
      new DragEventHandler(
        OnDragEnterDessertTopping);
    dessert.DragDrop +=
      new DragEventHandler(OnDropDessertTopping);

    TextBox textTarget = new TextBox();
```

```
    textTarget.Width = 400;
    textTarget.Location = new Point(120, 250);
    textTarget.AllowDrop = true;
    textTarget.DragEnter +=
      new DragEventHandler(OnDragEnterText);
    textTarget.DragDrop +=
      new DragEventHandler(OnDropText);

    Controls.AddRange(
      new Control[]{
        b, pb, floor, dessert, textTarget});
  }

  private void OnPutSuperGlo(object s, EventArgs a){
    SuperGlo superglo = new SuperGlo();
    SuperGloMover mover = new SuperGloMover();
    mover.SetData(superglo);
    Clipboard.SetDataObject(mover);
  }

  private void OnBeginDrag(
    object s, MouseEventArgs args) {
    SuperGloMover sgm = new SuperGloMover();
    sgm.SetData(new SuperGlo());
    ((Control) s).DoDragDrop(
      sgm, DragDropEffects.Copy );
  }

  private void OnDragEnterFloorWax(
    object s, DragEventArgs args) {
    if (args.Data.GetDataPresent("FloorWax")) {
      args.Effect = DragDropEffects.Copy;
    }
  }

  private void OnDropFloorWax (
    object s, DragEventArgs args) {
    FloorWax f =
      (FloorWax) args.Data.GetData("FloorWax");
    f.Polish();
  }
```

```
  private void OnDragEnterDessertTopping(
    object s, DragEventArgs args) {
    if (args.Data.GetDataPresent("DessertTopping")) {
      args.Effect = DragDropEffects.Copy;
    }
  }

  private void OnDropDessertTopping(
    object s, DragEventArgs args) {
    DessertTopping d =
      (DessertTopping) args.Data.GetData
      ("DessertTopping");
    d.Eat();
  }

  private void OnDragEnterText(
    object s, DragEventArgs args) {
    if (args.Data.GetDataPresent("Text")) {
      args.Effect = DragDropEffects.Copy;
    }
  }

  private void OnDropText(
    object sender, DragEventArgs args) {
    string s = (string) args.Data.GetData("Text");
    ((Control)sender).Text = s;
  }

  public static void Main() {
    Application.Run(new DragAndDropDemo());
  }
}///:~
```

When run, this application looks like this:

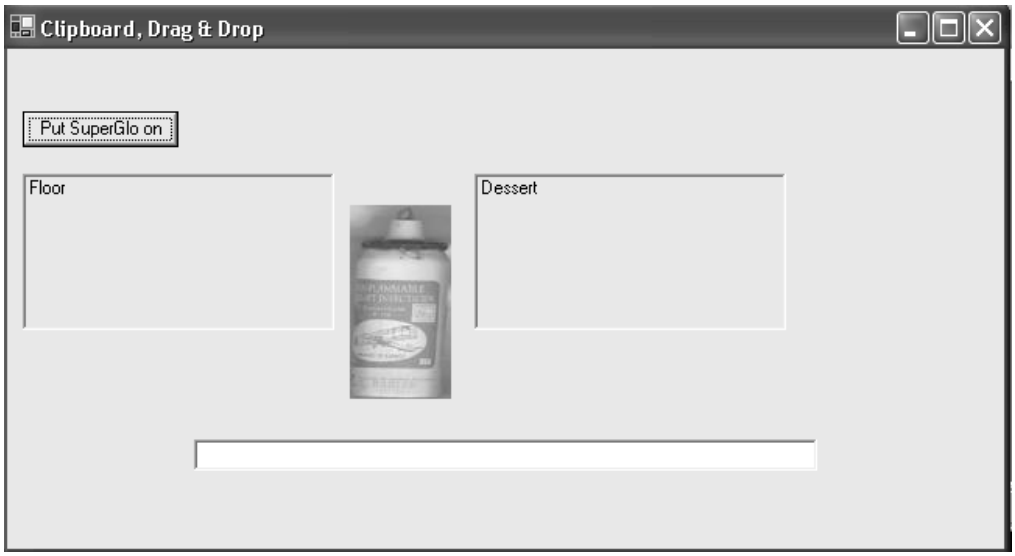


Figure 14-9: Dragging SuperGlo to different targets results in different behavior

If you click on the image, you can drag it to either of the two **Panel**s, the **TextEdit** display, or into a text editor such as Microsoft Word. As you drag, the mouse cursor will change to indicate whether or not the **SuperGlo** can be dropped. If you drop the **SuperGlo** on the "Floor" panel you'll see one message on the console, if on the "Dessert" panel another, and if in the **TextEdit** box or in another application, the promotional message will be pasted. If you click on the button, you can put a new **SuperGlo** on the Clipboard.

The **DragAndDropDemo( )** constructor should have no surprises until we add a **MouseEventHandler** on for the **MouseDown** event of the **PictureBox** control. That event will trigger the creation of a new **SuperGloMover** (as will the pressing of the **Button).**

The **Panel**s both have their **AllowDrop** property set to **true**, so when the mouse enters them while dragging an object, the mouse will give visual feedback that a drop target is being traversed. Additionally, the **Panel**'s **DragEnter** and **DragDrop** events are wired to event handlers.

When the button is pressed, it ends up calling **OnPutSuperGlo( )**. A new **SuperGlo** domain object is created, as is a new **SuperGloMover**. The **mover**'s data is set to the just-created **SuperGlo** and the **mover** is placed on the system Clipboard. In another application such as a text editor that accepts Clipboard text data, you should be able to paste data. That other application will end up calling **SuperGloMover.GetData( )** requesting **DataType.Text**, resulting in the

message “SuperGlo – It’s a FloorWax! And a dessert topping!” being pasted into the other application.

The method **OnBeginDrag( )** is pretty similar to **OnPutSuperGlo( )** but after the **SuperGloMover** is instantiated, the method **DoDragDrop( )** is called on the *originating* **Control**. This is somewhat confusing, as you might think that the destination should be in charge, but on reflection it makes sense that the originator knows the most about the data, even when the data is “far away” on the screen. There are several different **DragDropEffects**, which are bitwise combinable; in this case, the effect we want is to copy the data to the target.

Both **Panel**s have very similar **OnDragEnter*xxx*( )** methods. The **DragEventArgs args** has a **Data** property that contains the **SuperGloMover** created in **OnBeginDrag( ).** This **Data** is checked to make sure that it can present the appropriate type (“FloorWax” or “DessertTopping”) and if so, setting the **args.Effect** property makes the mouse show the appropriate visual cue for **DragDropEffects.Copy**.

Similarly, both **Panel**s have similar **OnDrop*xxx*( )** methods. The **Data** property (the **SuperGloMover**) has its **GetData( )** method called with the desired type, either a **FloorWax** or **DessertTopping**. This will return the **SuperGlo**. Either **FloorWax.Polish( )** or **DessertTopping.Eat( )** is called, and the appropriate message printed on the console.

The **TextBox** has similar **OnDragEnterText( )** and **OnDropText( )** methods, except the **SuperGloMover** is queried for the “Text” type.

Data-bound controls

Windows Forms and ADO.NET combine to make data-driven user interfaces very easy to program. In Chapter 10, we saw how ADO.NET separates the concerns of moving data in and out from a persistent datastore from the concerns of manipulating the in-memory **DataSet**. Once a **DataSet** is populated, instances of the **DataBinding** class mediate between **Control**s and **DataSet**s.

One of the more impressive **Control**s for displaying data is the **DataGrid**, which can display all the columns in a table. This example revisits the “Northwind.mdb” Access database:

```
//:c14:DataBoundDemo.cs
//Demonstrates the basics of data-binding
using System;
using System.Data;
```

```
using System.Data.OleDb;
using System.Windows.Forms;

class DataBoundDemo : Form {
  OleDbDataAdapter adapter;
  DataSet emps;

  DataBoundDemo(){
    DataGrid dg = new DataGrid();
    dg.Dock = DockStyle.Fill;
    Controls.Add(dg);

    ReadEmployees("NWind.mdb");

    dg.SetDataBinding(emps, "Table");
  }

  private void ReadEmployees(string pathToAccessDB){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString=
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + pathToAccessDB;
    cnctn.Open();

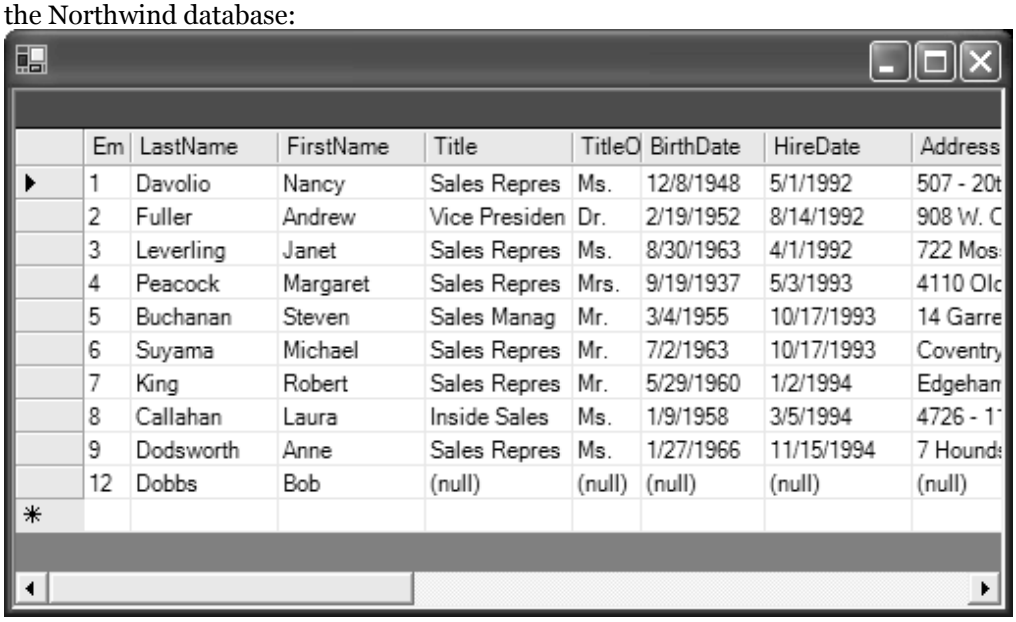
    string selStr = "SELECT * FROM EMPLOYEES";
    adapter = new OleDbDataAdapter(selStr, cnctn);
    new OleDbCommandBuilder(adapter);

    emps = new DataSet("Employees");
    adapter.Fill(emps);
  }

  public static void Main(){
    Application.Run(new DataBoundDemo());
  }
}///:~
```

A **DataGrid dg** control is created and set so that it will fill the entire client area of the **DataBoundDemo** form. Then, the **ReadEmployees( )** method fills the **DataSet emps**, just as in the examples in Chapter 10. Once **emps** is filled, the **dg.SetDataBinding( )** associates the **DataGrid** with the "Table" table in **emps**, which in this case is the only table in the **DataSet**. The result shows the contents of

the Northwind database:



	Em	LastName	FirstName	Title	TitleO	BirthDate	HireDate	Address
▶	1	Davolio	Nancy	Sales Repres	Ms.	12/8/1948	5/1/1992	507 - 20t
	2	Fuller	Andrew	Vice Presiden	Dr.	2/19/1952	8/14/1992	908 W. C
	3	Leverling	Janet	Sales Repres	Ms.	8/30/1963	4/1/1992	722 Mos
	4	Peacock	Margaret	Sales Repres	Mrs.	9/19/1937	5/3/1993	4110 Olc
	5	Buchanan	Steven	Sales Manag	Mr.	3/4/1955	10/17/1993	14 Garre
	6	Suyama	Michael	Sales Repres	Mr.	7/2/1963	10/17/1993	Coventry
	7	King	Robert	Sales Repres	Mr.	5/29/1960	1/2/1994	Edgehan
	8	Callahan	Laura	Inside Sales	Ms.	1/9/1958	3/5/1994	4726 - 1
	9	Dodsworth	Anne	Sales Repres	Ms.	1/27/1966	11/15/1994	7 Hound
	12	Dobbs	Bob	(null)	(null)	(null)	(null)	(null)
*								

Figure 14-10: Data access is easy with Windows Forms

Even more powerfully, the **BindingManagerBase** class can coordinate a set of **Binding**s, allowing you to have several **Control**s whose databound properties are simultaneously and transparently changed to correspond to a single record in the **DataSet**. This example allows you to navigate through the employee data:

```
//:c14:Corresponder.cs
//Demonstrates how bound controls can be made to match

using System;
using System.Drawing;
using System.Windows.Forms;
using System.Data;
using System.Data.OleDb;

class Corresponder : Form {
  OleDbDataAdapter adapter;
  DataSet emps;

  Corresponder(){
    ReadEmployees("NWind.mdb");
```

```
    Label fName = new Label();
    fName.Location = new Point(10, 10);
    Binding fBound =
      new Binding("Text", emps, "Table.FirstName");
    fName.DataBindings.Add(fBound);
    Controls.Add(fName);

    Label lName = new Label();
    lName.Location = new Point(10, 40);
    Binding lBound =
      new Binding("Text", emps, "Table.LastName");
    lName.DataBindings.Add(lBound);
    Controls.Add(lName);

    Button next = new Button();
    next.Location = new Point(100, 70);
    next.Text = ">";
    next.Click += new EventHandler(OnNext);
    Controls.Add(next);

    Button prev = new Button();
    prev.Location = new Point(10, 70);
    prev.Text = "<";
    prev.Click += new EventHandler(OnPrev);
    Controls.Add(prev);
  }

  void OnNext(object src, EventArgs ea){
    BindingManagerBase mgr =
      BindingContext[emps, "Table"];
    mgr.Position++;
  }

  void OnPrev(object src, EventArgs ea){
    BindingManagerBase mgr =
      BindingContext[emps, "Table"];
    mgr.Position--;
  }

  private void ReadEmployees(
    string pathToAccessDB){
```

```
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString=
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + pathToAccessDB;
    cnctn.Open();

    string selStr = "SELECT * FROM EMPLOYEES";
    adapter = new OleDbDataAdapter(selStr, cnctn);
    new OleDbCommandBuilder(adapter);

    emps = new DataSet("Employees");
    adapter.Fill(emps);
  }

  public static void Main(){
    Application.Run(new Corresponder());
  }
}///:~
```

After populating the **DataSet emps** from the database, we create a **Label** to hold the first name of the current employee. A new **Binding** object is created; the first parameter specifies the name of the Property to which the data will be bound (often, this will be the "Text" property), the second the **DataSet** that will be the data source, and the third the specific column in the **DataSet** to bind to. Once the **Binding** is created, adding it to **fName.Bindings** sets the **Label**'s **Text** property to the **DataSet** value.

The commented-out line immediately after **lName**'s constructor call is a call that will lead to a runtime exception – **Binding**s cannot be shared between **Control**s. If you wish multiple controls to reflect the same **DataSet** value, you have to create multiple **Binding**s, one for each **Control**.

A similar process is used to configure the **lName** label for the last name, and two **Button**s are created to provide rudimentary navigation.

The **Button**s' event-handlers use the **BindingContext**'s static **[ ]** operator overload which takes a **DataSet** and a **string** representing a data member. The data member in this case is the "Table" result. The resulting **BindingManagerBase.Property** value is then incremented or decremented within the event handler. As that happens, the **BindingManagerBase** updates its associated **Binding**s, which in turn update their bound **Control**s with the appropriate data from the **Binding**s' **DataSet.** This diagram illustrates the static structure of the relationships.

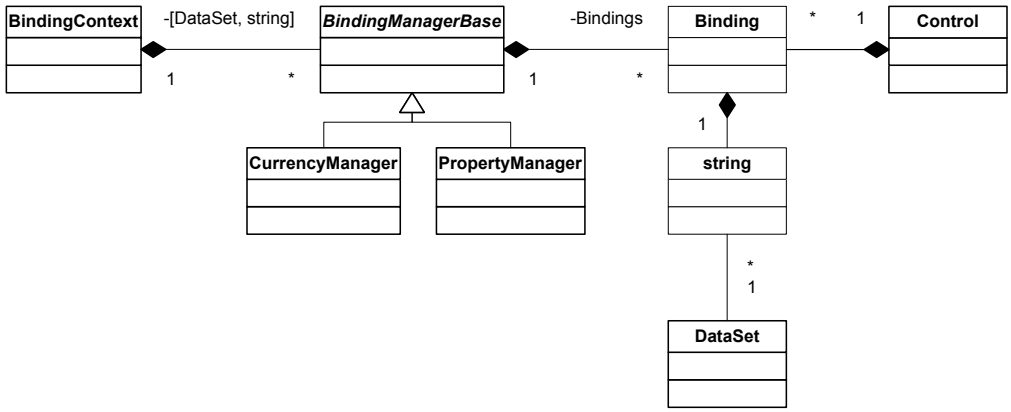


Figure 14-11: Coordinating the relationship between Controls and DataSets

You'll notice that the **BindingManagerBase** is an abstract class instantiated by **CurrencyManager** and **PropertyManager**. If the call to **BindingContext[object, string]** returns an **object** that implements **IList**, you'll get a **CurrencyManager** whose **Position** property moves back and forth between items in the **IList** (this is what was shown in the example; manipulating the **Position** in the event-handler switches between employees). If the call to **BindingContext[object, string]** does not return an object implementing **IList**, a **PropertyManager** is returned.

Editing data from bound controls

Objects of the **Binding** class manipulate the **DataSet**. As discussed in Chapter 10, changing the **DataSet** does not affect the backing store, the **IDataAdapter** is responsible for mediating between the in-memory **DataStore** and the **IDbConnection**.

This example demonstrates updating the database, provides a more detailed look at the events that occur during data binding, and illustrates a useful UI technique for manipulating databases or other large data structures. The UI technique is based on the premise that it may be expensive in terms of memory or resources or redrawing to have every editable portion of the screen actually *be* editable; rather, the **Control** that manages the editing process is dynamically positioned on the screen at the insertion point (or, in this case, when the mouse clicks on a data-bound **Label** which we wish to edit). This technique is not worth the effort on simple screens, but really comes into its own on complex UIs such as you might have on a word processor, spreadsheet, or a workflow application. If you use this technique, you should implement it using the PAC GUI architecture so that the result appears as a seamless component.

```
//:c14:FormEdit.cs
//Demonstrates data write from Form
using System;
using System.Collections;
using System.Data;
using System.Data.OleDb;
using System.Drawing;
using System.Windows.Forms;

class FormEdit : Form {
  Hashtable colForLabel = new Hashtable();
  bool editWindowActive = false;
  TextBox editWindow = new TextBox();
  Label labelBeingEdited = null;

  OleDbDataAdapter adapter;
  DataSet emps;

  FormEdit(){
    ReadEmployees("NWind.mdb");
    InitDataStructure();
    InitLabels();
    InitCommitter();
  }

  private void InitDataStructure(){
    colForLabel[new Label()] = "FirstName";
    colForLabel[new Label()] = "LastName";
  }

  private void InitLabels(){
    int x = 10;
    int y = 10;
    int yIncrement = 35;

    foreach(string colName in colForLabel.Values){
      //Bihashtable w KeyForValue() not in library
      foreach(Label lbl in colForLabel.Keys){
        string aColName =
          (string) colForLabel[lbl];
        if (aColName == colName) {
```

```
        //Right key (label) for value (colName)
        InitLabel(lbl, colName, x, y);
        y += yIncrement;
      }
    }
  }
}

private void InitLabel(Label lbl, string colName,
  int x, int y){
  lbl.Location = new Point(x, y);
  lbl.Click += new EventHandler(OnLabelClick);
  lbl.TextChanged +=
    new EventHandler(OnTextChange);
  Controls.Add(lbl);

  string navPath = "Table." + colName;
  Binding b =
    new Binding("Text", emps, navPath);
  b.Parse +=
    new ConvertEventHandler(OnBoundDataChange);
  lbl.DataBindings.Add(b);
}

private void InitCommitter(){
  Button b = new Button();
  b.Width = 120;
  b.Location = new Point(150, 10);
  b.Text = "Commit changes";
  b.Click += new EventHandler(OnCommit);
  Controls.Add(b);
}

public void OnLabelClick(Object src, EventArgs ea){
  Label srcLabel = (Label) src;
  if (editWindowActive)
    FinalizeEdit();
  PlaceEditWindowOverLabel(srcLabel);
  AssociateEditorWithLabel(srcLabel);
}
```

```
private void PlaceEditWindowOverLabel(Label lbl){
  editWindow.Location = lbl.Location;
  editWindow.Size = lbl.Size;
  if (Controls.Contains(editWindow) == false) {
    Controls.Add(editWindow);
  }
  editWindow.Visible = true;
  editWindow.BringToFront();
  editWindow.Focus();
  editWindowActive = true;
}

private void AssociateEditorWithLabel(Label l){
  editWindow.Text = l.Text;
  labelBeingEdited = l;
}

public void FinalizeEdit(){
  Console.WriteLine("Finalizing edit");
  labelBeingEdited.Text = editWindow.Text;
  Console.WriteLine("Text changed");
  //Needed to trigger binding event
  labelBeingEdited.Focus();

  editWindow.Visible = false;
  editWindow.SendToBack();
  editWindowActive = false;
}

public void OnTextChange(Object src, EventArgs ea){
  Label lbl = (Label) src;
  string colName = (string) colForLabel[lbl];
  Console.WriteLine(
    colName + " has changed");
}

public void OnBoundDataChange(
  Object src, ConvertEventArgs ea){
  Console.WriteLine("Bound data changed");
}
```

```
  private void OnCommit(Object src, EventArgs ea){
    FinalizeEdit();
    adapter.Update(emps);
  }

  private void ReadEmployees(
    string pathToAccessDB){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString =
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + pathToAccessDB;
    cnctn.Open();

    string selStr = "SELECT * FROM EMPLOYEES";
    adapter = new OleDbDataAdapter(selStr, cnctn);
    new OleDbCommandBuilder(adapter);

    emps = new DataSet("Employees");
    adapter.Fill(emps);
  }

  public static void Main(){
    Application.Run(new FormEdit());
```

This example is very clean and easy to follow.

```
  }
}///:~
```

The **FormEdit** class has several instance variables: **colForLabel** is a **Hashtable** that is used to establish a correspondence between a **DataSet** column name and a **Label** control. The next three instance variables, **editWindowActive**, **editWindow**, and **labelBeingEdited** are used in the "dynamic" editing scheme – all editing is done in the **editWindow TextBox**, while a reference to the editing target is held in the **labelBeingEdited** variable. The **adapter** and **emps** variables are the familiar variables holding the **IDataAdapter** and **DataSet** associated with the Northwind database.

The **FormEdit( )** constructor initializes **emps** and **adapter** in **ReadEmployees( )**, which does not differ from the previous examples. **InitDataStructure( )** initializes the **colForLabel** data structure; an anonymous **Label** is used as the key and the column's name is the value. One can

imagine a more sophisticated version of this method that iterates over the **DataSet**, creating as many **Control**s as are necessary.

InitLabels( ) is responsible for setting up the display of the data structure initialized in **InitDataStructure( )**. Because the .NET Framework does not have a bidirectional version of the **IDictionary** interface that can look up a key based on the value, we have to use a nested loop to find the **Label** associated with the **colName**. Once found, **InitLabels( )** calls **InitLabel( )** to configure the specific **Label**. Again, a more sophisticated version of this method might do a more sophisticated job of laying out the display.

InitLabel( ) is responsible for wiring up the **Label** to the **DataSet**'s column name. Two event handlers are added to the **Label**: **OnLabelClick( )** and **OnTextChange( )**. The method also associates **OnBoundDataChange( )** with the **Binding**'s **Parse** event, which occurs when the value of a databound control changes. The **Binding** is created from the column name that was passed in as an argument and associated with the **Label** that was also passed in.

The last method called by the constructor is **InitCommitter( ),** which initializes a **Button** that will be used to trigger a database update.

After the constructor runs, the **DataSet emps** is filled with Northwind's employee data and the **Label**s on the **EditForm** show the first and last name of the first record (this example doesn't have any navigation). When the user clicks on one of these labels, they activate **OnLabelClick( )**.

All **Label**s on the form share the **OnLabelClick( )** event handler, but simply casting the **src** to **Label** gives us the needed information to manipulate the **editWindow**. The first time through **OnLabelClick( )**, the **editWindowActive** Boolean will be **false**, so let's momentarily delay discussion of the **FinalizeEdit( )** method.

Our dynamic editing technique is to place the editing control on the UI at the appropriate place, and then to associate the editing control with the underlying data. **PlaceEditWindowOverLabel( )** sets the **editWindow** to overlay the **Label** the user clicked on, makes it visible, and requests the focus. **AssociateEditorWithLabel( )** sets the text of the **editorWindow** to what is in the underlying **Label** and sets the **labelBeingEdited** instance variable to the clicked-on **Label**. The visual effect of all this is that when someone clicks on either **Label**, it appears as if the **Label** turns into an **EditBox**. Again, this is no big deal with two labels on the form, but if there were a couple dozen fields beings displayed on the screen, there can be a significant resource and speed improvement by having a single, dynamic editing control.

Since **PlaceEditWindowOverLabel( )** set **editWindowActive** to **true**, subsequent calls to **OnLabelClick( )** will call **FinalizeEdit( )**. **FinalizeEdit( )** puts the text from the **editWindow** into the underlying **labelBeingEdited**. This alone *does not update* the **DataBinding**, you must give the label the focus, even for a moment, in order to update the data. **OnBoundDataChanged( )** is called subsequent to the data update, and when run, you'll see:

```
Finalizing edit
FirstName has changed
Text changed
Bound data changed
```

Indicating this sequence:

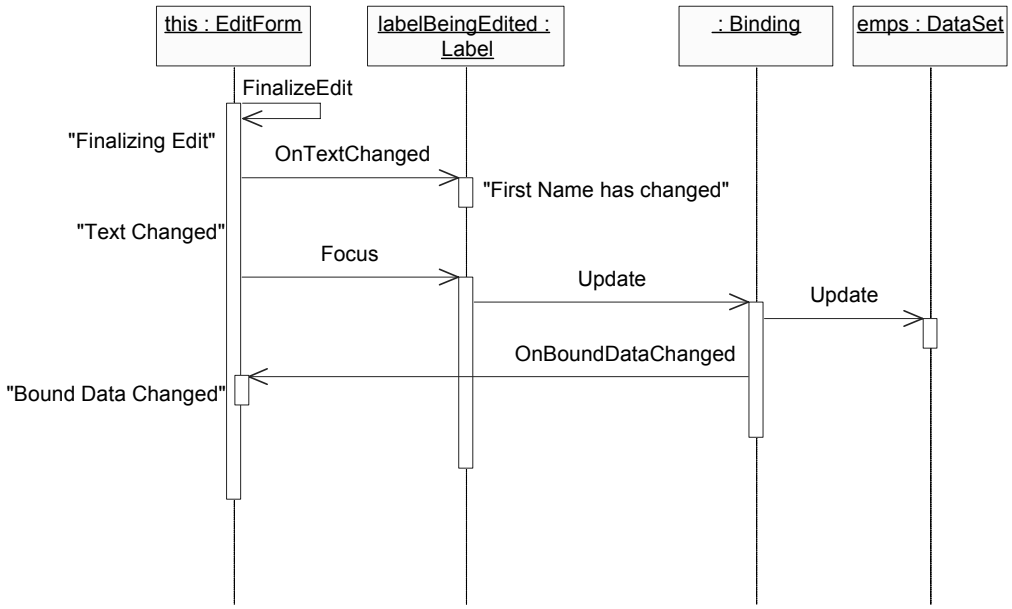


Figure 14-12: The sequence by which a data change propagates

The above diagram is a UML *activity diagram* and is one of the most helpful diagrams for whiteboard sessions (as argued effectively by Scott Ambler at *http://www.agilemodeling.org*, you will derive more benefit from low-tech modeling with others than you can ever derive from high-tech modeling by yourself).

The activity diagram has time on the vertical axis and messages and objects on the horizontal. The boxes on the vertical lines correspond to method invocations

and the call stack. Thus, **this.OnBoundDataChanged( )** is called from an anonymous **BindingObject**'s **Update( )**[1] method, which is called from **labelBeingEdited.Focus( ),** which we call in **this.FinalizeEdit( )**.

Although **FinalizeEdit( )** modifies the **DataSet emps**, the call to **adapter.Update( )** in **OnCommit( )** is required to actually move the data back into the database. Be sure to be familiar with ADO.NET's optimistic concurrency behavior, as discussed in Chapter 10, before engaging in any database projects.

Menus

User interface experts say that menus are overrated. Programmers, who are trained to think in terms of hierarchies, and whose careers are computer-centric, don't bat an eye at cascading menu options and fondly recall the DOS days when you could specify incredibly complex spreadsheet and word processing behaviors by rattling off the first letters of menu sequences (nowadays, you have to say "So are you seeing a dialog that says 'Display Properties' and that has a bunch of tabs? Okay, find the tab that says 'Appearance' and click on it. Do you see the button that reads 'Advanced'? It should be in the lower-right corner…"). For many users, though, menus are surprisingly difficult to navigate. Nevertheless, unless you're creating a purely Web-based interface, the odds are quite good that you'll want to add menus and context menus to your UIs.

There are two types of menu in Windows Forms, a **MainMenu** that displays across the top of a **Form**, and a **ContextMenu** that can be associated with any **Control** and is typically displayed on a right-button mouse click over the **Control**.

Both forms of menu contain **MenuItem**s and **MenuItem**s may, in turn, contain other **MenuItem**s (once again, the familiar containment model of Windows Forms) . When selected, the **MenuItem** triggers a **Click** event, even if the **MenuItem** was chosen via a shortcut or access key. This example shows both types of menus and how to create cascading menus:

```
//:c14:MenuDemo.cs
//Demonstrates menus
```

[1] Actually, the names and sequence of events that happen in **Control.Focus( )** is considerably more complex than what is portrayed here, but glossing over details not relevant to the task at hand is one of the great benefits of diagramming. This is one of the reasons diagramming tools that dynamically bind to actual source code are often less useful than, say, a piece of paper.

```
using System;
using System.Drawing;
using System.Windows.Forms;

class MenuDemo : Form {
  MenuDemo(){
    Text = "Menu Demo";
    MainMenu courseMenu = new MainMenu();

    MenuItem appetizers = new MenuItem();
    appetizers.Text = "&Appetizers";
    MenuItem[] starters = new MenuItem[3];
    starters[0] = new MenuItem();
    starters[0].Text = "&Pot stickers";
    starters[1] = new MenuItem();
    starters[1].Text = "&Spring rolls";
    starters[2] = new MenuItem();
    //Note escaped "&"
    starters[2].Text = "&Hot && Sour Soup";
    appetizers.MenuItems.AddRange(starters);
    foreach(MenuItem i in starters){
      i.Click +=
        new EventHandler(OnCombinableMenuSelected);
    }

    MenuItem mainCourse = new MenuItem();

    mainCourse.Text = "&Main Course";
    MenuItem[] main = new MenuItem[4];
    main[0] = new MenuItem();
    main[0].Text = "&Sweet && Sour Pork";
    main[1] = new MenuItem();
    main[1].Text = "&Moo shu";
    main[2] = new MenuItem();
    main[2].Text = "&Kung Pao Chicken";
    //Out of Kung Pao Chicken
    main[2].Enabled = false;
    main[3] = new MenuItem();
    main[3].Text = "General's Chicken";
    mainCourse.MenuItems.AddRange(main);
```

```
    foreach(MenuItem i in main){
      i.RadioCheck = true;
      i.Click +=
        new EventHandler(OnExclusiveMenuSelected);
    }

    MenuItem veg = new MenuItem();
    veg.Text = "Vegetarian";
    veg.RadioCheck = true;
    veg.Click +=
      new EventHandler(OnExclusiveMenuSelected);
    MenuItem pork = new MenuItem();
    pork.Text = "Pork";
    pork.RadioCheck = true;
    pork.Click +=
      new EventHandler(OnExclusiveMenuSelected);
    main[1].MenuItems.AddRange(
      new MenuItem[]{veg,pork});

    courseMenu.MenuItems.Add(appetizers);
    courseMenu.MenuItems.Add(mainCourse);

    ContextMenu contextMenu = new ContextMenu();
    foreach(MenuItem a in starters){
      contextMenu.MenuItems.Add(a.CloneMenu());
    }
    contextMenu.MenuItems.Add(
      new MenuItem().Text = "-");
    foreach(MenuItem m in main){
      contextMenu.MenuItems.Add(m.CloneMenu());
    }
    Menu = courseMenu;
    ContextMenu = contextMenu;
  }

  private void OnCombinableMenuSelected(
    object sender, EventArgs args){
    MenuItem selection = (MenuItem) sender;
    selection.Checked = !selection.Checked;
  }
```

```
  private void OnExclusiveMenuSelected(
    object sender, EventArgs args){
    MenuItem selection = (MenuItem) sender;
    bool selectAfterClear = !selection.Checked;
    //Must implement radio-button functionality
    //programmatically
    Menu parent = selection.Parent;
    foreach(MenuItem i in parent.MenuItems){
      i.Checked = false;
    }
    selection.Checked = selectAfterClear;
  }

  public static void Main(){
    Application.Run(new MenuDemo());
  }
}///:~
```

The **MenuDemo( )** constructor creates a series **MenuItem**s. The ampersand (&) in the **MenuItem.Text** property sets the accelerator key for the item (so "Alt-A" will activate the "Appetizers" menu and "Alt-M" the "Main Course" menu). To include an ampersand in the menu text, you must use a double ampersand (for instance, "**Hot && Sour Soup**").

Each **MenuItem** created is added to either the **mainMenu**.**MenuItems** collection or to the **MenuItems** collection of one of the other **MenuItem**s.

The **MenuItem**s in the **appetizers Menu** have the default **false** value for their **RadioCheck** property. This means that if their **Selection.Checked** property is set to **true**, they will display a small checkmark. The **MenuItem**s in **mainCourse** have **RadioCheck** set to **true**, and when they have **Selected.Checked** set to true, they display a small circle. However, this is a display option only, the mutual exclusion logic of a radio button must be implemented by the programmer, as is shown in the **OnExclusiveMenuSelected( )** method.

Additionally, **RadioCheck** does not "cascade" down into child menus. So, in this case, it's possible to select two main courses if one of the selected main courses is in the "Moo Shu" sub-menu.

A **MenuItem** is deactivated by setting its **Enabled** property to false, as is done with the "Kung Pao Chicken" entrée.

To duplicate a **Menu,** you use not **Clone( )** but **CloneMenu( )**, as shown in the loops that populate the **contextMenu**. The **contextMenu** also demonstrates that a **MenuItem** with its **Text** property set to a single dash is displayed as a separator bar in the resulting menu.

Standard dialogs

Windows Forms provides several standard dialogs both to ease common chores and maintain consistency for the end user. These dialogs include file open and save, color choice, font dialogs, and print preview and print dialogs. We're going to hold off on the discussion of the printing dialogs until the section on printing in the GDI+ chapter, but this example shows the ease with which the others are used:

```
//:c14:StdDialogs.cs
using System;
using System.Drawing;
using System.Windows.Forms;

class StdDialogs : Form {
  Label label = new Label();

  StdDialogs(){
    MainMenu menu = new MainMenu();
    Menu = menu;

    MenuItem fMenu = new MenuItem("&File");
    menu.MenuItems.Add(fMenu);

    MenuItem oMenu = new MenuItem("&Open...");
    oMenu.Click += new EventHandler(OnOpen);
    fMenu.MenuItems.Add(oMenu);

    MenuItem cMenu = new MenuItem("&Save...");
    cMenu.Click += new EventHandler(OnClose);
    fMenu.MenuItems.Add(cMenu);
    fMenu.MenuItems.Add(new MenuItem("-"));

    MenuItem opMenu = new MenuItem("&Options");
    menu.MenuItems.Add(opMenu);

    MenuItem clrMenu = new MenuItem("&Color...");
```

```
    clrMenu.Click += new EventHandler(OnColor);
    opMenu.MenuItems.Add(clrMenu);

    MenuItem fntMenu = new MenuItem("&Font...");
    fntMenu.Click += new EventHandler(OnFont);
    opMenu.MenuItems.Add(fntMenu);

    label.Text = "Some text";
    label.Dock = DockStyle.Fill;
    Controls.Add(label);
  }

  public void OnOpen(object src, EventArgs ea){
    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Filter =
      "C# files (*.cs)|*.cs|All files (*.*)|*.*";
    ofd.FilterIndex = 2;

    DialogResult fileChosen = ofd.ShowDialog();
    if (fileChosen == DialogResult.OK) {
      foreach(string fName in ofd.FileNames){
        Console.WriteLine(fName);
      }
    } else {
      Console.WriteLine("No file chosen");
    }
  }

  public void OnClose(object src, EventArgs ea){
    SaveFileDialog sfd = new SaveFileDialog();
    DialogResult saveChosen = sfd.ShowDialog();
    if (saveChosen == DialogResult.OK) {
      Console.WriteLine(sfd.FileName);
    }
  }

  public void OnColor(object src, EventArgs ea){
    ColorDialog cd = new ColorDialog();
    if (cd.ShowDialog() == DialogResult.OK) {
      Color c = cd.Color;
      label.ForeColor = c;
```

```
      Update();
    }
  }

  public void OnFont(object src, EventArgs ea){
    FontDialog fd = new FontDialog();
    if (fd.ShowDialog() == DialogResult.OK) {
      Font f = fd.Font;
      label.Font = f;
      Update();
    }
  }

  public static void Main(){
    Application.Run(new StdDialogs());
  }
}///:~
```

The **StdDialogs( )** constructor initializes a menu structure and places a **Label** on the form. **OnOpen( )** creates a new **OpenFileDialog** and sets its **Filter** property to show either all files or just those with **.cs** extensions. File dialog filters are confusing. They are specified with a long string, delimited with the pipe symbol (|). The displayed names of the filters are placed in the odd positions, while the filters themselves are placed in the even positions. The **FilterIndex** is *one-based!* So by setting its value to 2, we're settings its value to the *.* filter, the fourth item in the list.

Like all the dialogs, the **OpenFileDialog** is shown with the **ShowDialog( )** method. This opens the dialog in *modal* form; the user must close the modal dialog before returning to any other kind of input to the system. When the dialog is closed, it returns a value from the **DialogResult** enumeration. **DialogResult.OK** is the hoped-for value; others are:

- Abort
- Cancel
- Ignore
- No
- None
- Retry
- Yes

Obviously, not all dialogs will return all (or most) of these values.

Both the **OpenFileDialog** and the **SaveFileDialog** have **OpenFile( )** methods that open the specified file (for reading or writing, respectively), but in the example, we just use the **FileNames** and **FileName** properties to get the list of selected files (in **OnOpen( )**) and the specified file to be written to in **OnSave( )**. The **SaveFileDialog.OverwritePrompt** property, which defaults to **true**, specifies whether the dialog will automatically ask the user "Are you sure…?"

OnColor( ) and **OnFont( )** both have appropriate properties (**Color** and **Font**) that can be read after the dialogs close. The **label** is changed to use that value and then the **Update( )** method is called in order to redraw and relayout the **StdDialogs** form.

Usage-centered design

In the discussion of GUI Architectures, we mentioned the adage that "To the end user, the UI *is* the application." This is one of the tenets of *usage-centered design*, an approach to system development that fits very well with the move towards *agile development* that propose that shorter product cycles (as short as a few weeks for internal projects) that are intensely focused on delivering requested end-user value are much more successful than the traditional approach of creating a balance of functional, non-functional, and market-driven features that are released in "major roll-outs."

Far too many discussions of software development fail to include the end user, much less accord them the appropriate respect as the driver of what the application should do and how it should appear. Imagine that one day you were given a survey on your eating habits as part of a major initiative to provide you a world-class diet. Eighteen months later, when you'd forgotten about the whole thing, you're given a specific breakfast, lunch, and dinner and told that this was all you could eat for the next eighteen months. And that the meals were prepared by dieticians, not chefs, and that the meals had never actually been tasted by anyone. That's how most software is developed.

No interface library is enough to make a usable interface. You *must* actively work with end users to discover their needs (you'll often help *them* discover the words to express a need they assumed could not be fixed), you *must* silently watch end users using your software (a humbling experience), and you *must* evolve the user

interface based on the needs of the user, not the aesthetic whims of a graphic designer.[2]

One of us (Larry) had the opportunity to work on a project that used usage-centered design to create new types of classroom management tools for K-12 teachers (a group that is rightfully distrustful of the technical "golden bullets" that are regularly foisted upon them). The UI had lots of standard components, and three custom controls. Two of the custom controls were *never noticed* by end users (they just used them, not realizing that they were seeing complex, non-standard behavior). We could always tell when users saw the third, though, because they literally *gasped* when they saw it. It was a heck of a good interface.[3]

Summary

C# has an object-oriented bound method type called *delegate*. Delegates are first-class types, and can be instantiated by any method whose signature exactly matches the delegate type.

There are several architectures that may be used to structure the GUI, either as an independent subsystem or for the program as a whole. The Visual Designer tool in Visual Studio .NET facilitates an architecture called Form-Event-Control, which provides a minimal separation of the interface from the domain logic. Model-View-Controller provides the maximum separation of interface, input, and domain logic, but is often more trouble than it's worth. Presentation-Abstraction-Control is an architecture that creates self-contained components that are responsible for both their own display and domain logic; it is often the best choice for working in Windows Forms.

Windows Forms is a well-architected class library that simplifies the creation of the vast majority of user interfaces. It is programmed by a series of public delegate properties called *events* that are associated with individual **Control**s. **Control**s contain other **Control**s. A **Form** is a type of **Control** that takes the form of a window.

---

[2] This is not to disparage graphic designers or their work. But creating a usable interface is *nothing* like creating a readable page or an eye-catching advertisement. If you can't get a designer with a background in Computer-Human Interaction (CHI), at least use a designer with a background in industrial design.

[3] Unfortunately, the CEO saw fit to spend our $15,000,000 in venture capital on prostitutes, drugs, and a Jaguar sedan with satellite navigation for the company car. Oh yeah, we also had free soda.

Controls are laid out within their containing **Control** by means of the **Location**, **Dock**, and **Anchor** properties. This layout model is simple, but not simplistic, and if combined with a proper attention to GUI architecture, can lead to easily modified, easy-to-use user interfaces.

Properties of **Control**s can be bound to values within a **DataSet** via a collection of **Binding**s. Often, the bound property is the "Text" property of the **Control**. A collection of **Binding**s may be coordinated by a **BindingManagerBase** provided by a **BindingContext**. Such coordinated **Binding**s will refer to a single data record or set of properties and thus, **Control**s bound to those **Binding**s will simultaneously update.

While Windows Forms is well-architected, there are many quirks and inconsistencies in the various **Control**s. These quirks range from what are clearly defects (**CheckedListBox.CheckedIndexCollection** includes items whose checkstate is **indeterminate**), to design flaws (there should be a **Control.Tooltip** property), to undocumented behaviors (when handed an object of illegal type, should **IDataObject.SetObject( )** throw an exception?), to behaviors that reflect an underlying quirk in the operating system (the string that specifies **OpenFileDialog.Filter**).

Visual Studio .NET makes the creation of Windows Forms interface very easy but is no substitute for working directly with the implementing code. Most real applications will use a combination of Visual Designer-generated code and hand-written implementations. One way or the other, the success of your application is dependent on the usability of your interface; if at all possible, work with a computer-human interface specialist to guide the creation of your UI.

Exercises

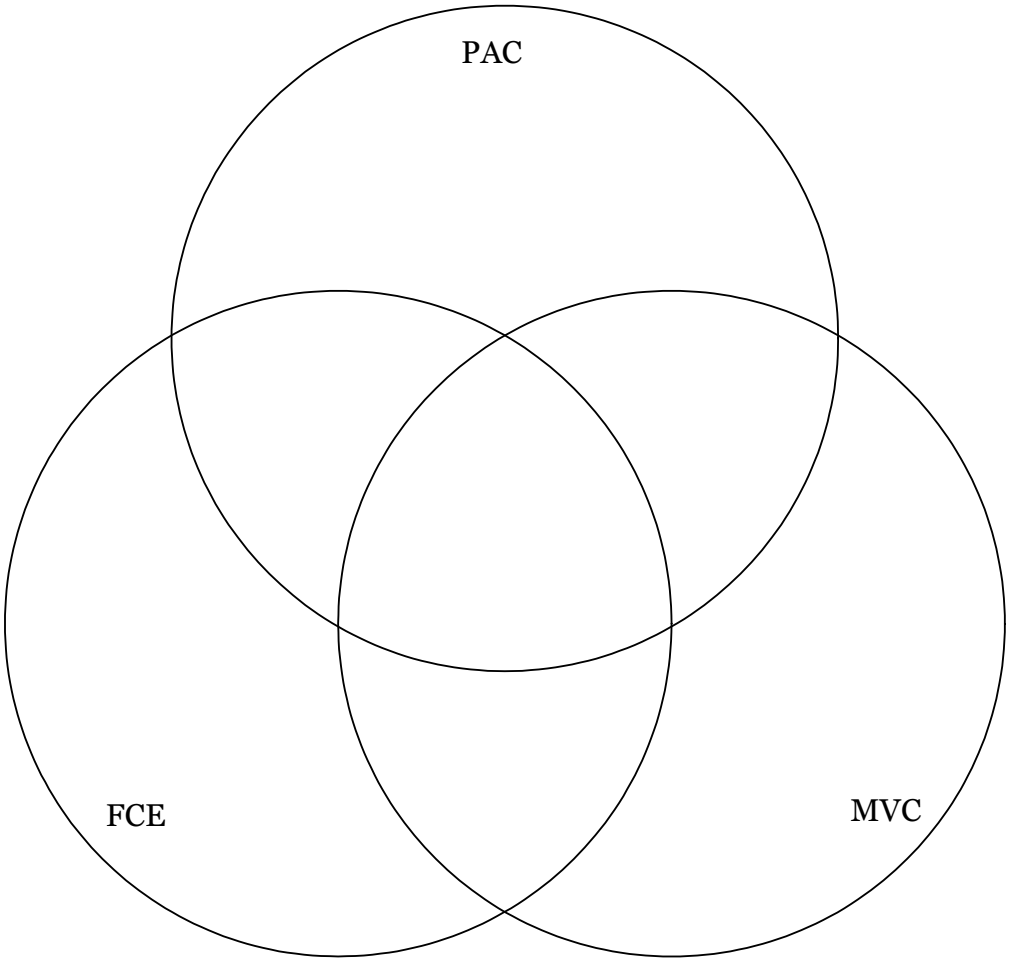
1. Extend **Profession.cs** with a new profession and a new method. Instantiate the **Profession** delegate with this new data.

2. Refactor the previous example so that your **Profession** delegate is instantiated by a method in a different class.

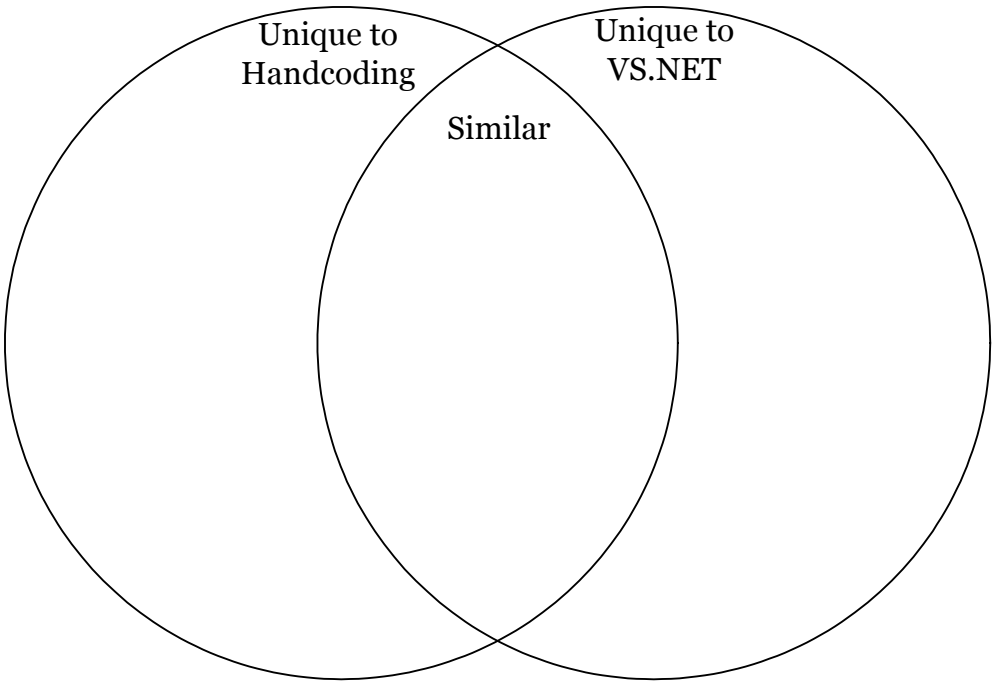
3. Add a **GarbageTruck** class that performs **CollectGarbage( )** as part of the morning routine in **Multicast.cs**.

4. Add to **EventProperty.cs** a **Bird** class that performs **CatchTheWorm( )** if the dawn is rainy and **Sing( )**s if the day is fair.

5. Fill in this Venn diagram with aspects of the three GUI architectures described in this chapter:



6. Implement at least three of the programs in this chapter using Visual Studio .NET's visual designer. Fill in this Venn diagram comparing aspects of hand coding a UI versus using this tool.



7. Add a new language to **International.cs**.

8. Write a program that shows the various tables in the **Northwind** database using a **ListView** in one pane and a **DataGrid** in another.

9. Write an interface that compares to Microsoft Outlook: a menu, a treeview in a left-hand pane, a list (or **ListView**) in an upper-right pane, and a detail view in a lower-right pane. Use it to, say, organize your digital medias.

15: GDI+ Overview

While Windows Forms provides a great basis for the large majority of user interfaces, the .NET Framework allows access to the full rendering capabilities of Windows XP. Windows Forms interfaces are based on the concept of **Control**s that, among other things, know how to draw themselves. If your interface requires drawing that's beyond the capabilities of the **Control**s at your disposal, you'll need to turn to .NET's GDI+ namespaces.

GDI+ provides a range of drawing, coordinate, and measurement tools ranging from simple line-drawing to complex gradient fills. The advantage of this is that virtually any kind of interface can be created using GDI+ (3D interfaces require DirectX, which will be discussed later). The disadvantage is that GDI+ is stateless and you must write code capable of re-rendering the *entire GDI+ interface* at any time. Most GDI+ work will also involve writing custom input code.

The amount of detail involved in handling redraws and input means that you must pay even more attention to separating domain logic from your interface code. It's difficult for the FEC architecture to handle the complexity of a GDI+ interface. PAC should still be where the discussion begins, but the power of MVC can become more attractive as one contemplates building UIs with innovative display or input characteristics. The sample code in this chapter does *not* separate domain logic from display and should not be used as a starting place for your designs.

Your canvas: the Graphics Class

The **Control** class is at the center of Windows Forms programming; you place a **Control**, you set certain attributes of it, you associate it with business logic. All of these hold true in GDI+ programs, except that you will be responsible for drawing *everything* within the client area of your control. Typicallly, you will create a new class inheriting from **Panel**, and define your own properties to control your object's appearance. You'll sometimes hear people referring to this process as developing an *owner-draw control***.**

The canvas on which you draw is an instance of the **Graphics** class. This class encapsulates the GDI+ drawing surface *for your **Control***. You do not have to worry about other windows (or even other **Control**s), screen location, and so forth. You can still use properties such as **Dock**, **Anchor**, and **Position** to handle the task of placing your custom **Control** within a general Windows Forms interface.

Every instance of **Graphics** that you use consumes a low-level operating system resource (a Win32 handle). This leads to two restrictions:

- You must always call **Dispose( )** on a **Graphics** object when you are done with it; you can either do this in a **try...finally** block or with the **using** keyword.
- You must not maintain a reference to a **Graphics( )** object outside of the event handler which obtained it, as the underlying handle is not guaranteed to be valid over time.

There are several ways to obtain a reference to a **Graphics** object. The most direct is to call **Control.CreateGraphics( )**, a method whose name highlights the transient nature of the resulting object. This example places two buttons on a **Form**. When the **controller** is clicked, it gets a reference to a **Graphics** object for the **target** and fills the **target**'s client area with red.

```
//:c15:GraphicsHandle.cs
//Accessing the drawing surface of a control
using System;
using System.Drawing;
using System.Windows.Forms;

class GraphicsHandle : Form {
  Button target;

  GraphicsHandle(){
    target = new Button();
    target.Location = new Point(10, 10);
    Controls.Add(target);

    Button controller = new Button();
    controller.Location = new Point(10, 60);
    controller.Text = "Clear target Graphics";
    controller.Width = 150;
    controller.Click += new EventHandler(OnClick);
```

```
    Controls.Add(controller);
  }

  public void OnClick(object src, EventArgs ea){
    Graphics canvas = target.CreateGraphics();
    using(canvas){
      canvas.Clear(Color.Red);
    }
  }

  public static void Main(){
    Application.Run(new GraphicsHandle());
  }
}///:~
```

After **OnClick( )** calls **target.CreateGraphics( ),** it wraps the use of the resulting object in **using**, which as discussed in Chapter 11 expands behind-the-scenes into a **try...finally** block that calls **Dispose( )** on its **IDisposable** argument.

Understanding repaints

The **GraphicsHandle** sample can illustrate some Windows behavior that can be confusing. Run the program and press the "Clear target Graphics" button. The **target** button will disappear, replaced by a red rectangle. Now minimize or otherwise obscure the **GraphicsHandle** application and then uncover it. The red rectangle is now replaced by the appearance of the normal button. So far, this seems logical: When the **target** button is redrawn, it draws itself as a button, when the **controller** button is clicked, the **Clear(Color.Red)** call temporarily replaces the button's "real" appearance.

Now press "Clear target Graphics" and move the application window around the screen; the red rectangle remains. This might make you go "Hmm...," since moving a window involves turning pixels on and off, i.e., repainting. Why doesn't the button redraw itself in its normal way?

Now do something that *partially* obscures the red rectangle (move a window edge over the control, or move the **GraphicsHandle** demo off the edge of the screen) and then uncover it. Now you'll see that the portion of the **target** button that was obscured gets repainted as a normal button, while the portion that was not obscured remains a red rectangle. What's going on?

The answer lies in the underlying Windows system for controlling the display. Essentially, Windows *tries* to avoid asking for a repaint. If the top-level window is being moved, Windows doesn't ask for a repaint at all, it just moves the pixels in the display card's memory. If a window is partially obscured and then revealed, Windows only repaints the affected area. If Windows used a different architecture, in which the entire client area was repainted, applications would show noticeable flicker even on fast machines.

This underlying argues strongly for *not* grabbing another **Control**s **Graphics** context, drawing on it, and then disposing it; any drawing that you do in this manner is, as shown in the **GraphicsHandle** demo, temporary.

Control**: paint thyself**

In Windows Forms, the **Paint** event triggers the redrawing of the client area. All **Control**s have a **protected OnPaint( )** method which is responsible for rendering. This is the preferred method for creating an owner-drawn control – inherit from an existing control and override **OnPaint( )**. This example shows a custom **Panel** that draws a sine wave from individual pixels.

```
//:c15:SineWave.cs
//Demonstrates GDI+ Drawing
using System;
using System.Drawing;
using System.Windows.Forms;

class OwnerDrawPanel : Panel {
  internal OwnerDrawPanel(){
    ResizeRedraw = true;
  }

  Color c = Color.Blue;
  static int drawCount = 0;

  protected override void OnPaint(PaintEventArgs e){
    base.OnPaint(e);
    Console.WriteLine("PaintSine called");
    Graphics g = e.Graphics;
    g.Clear(Color.White);
    Pen pen = null;
    if (drawCount == 0) {
      pen = new Pen(Color.Blue);
```

```
    } else {
      pen = new Pen(Color.Red);
    }
    drawCount++;
    double inc = Math.PI * 4 / Width;
    int x = 0;
    for (double d = 0; d < Math.PI * 4; d += inc) {
      double sin = Math.Sin(d);
      int y = (int) (this.Height / 2 * sin);
      y += this.Height / 2;
      Rectangle rec = new Rectangle(x, y, 1, 1);
      g.DrawRectangle(pen,rec);
      x++;
    }
  }
}

class SineWave : Form {
  SineWave(){
    Panel p = new Panel();
    p.Dock = DockStyle.Left;
    p.Width = 120;

    Splitter s = new Splitter();
    s.Dock = DockStyle.Left;
    Controls.Add(s);
    Controls.Add(p);

    OwnerDrawPanel ownerDraw = new OwnerDrawPanel();
    ownerDraw.Dock = DockStyle.Fill;
    Controls.Add(ownerDraw);
  }

  public static void Main(){
    Application.Run(new SineWave());
  }
}///:~
```

The **OwnerDrawPanel( )** constructor specifies that the **ResizeRedraw** property inherited from **Control** is **true**. This should be set to **true** if, as in this case, the control needs to redraw its entire client area on a resize event. The

downside to setting this property to **true** is that the **Control** is much more likely to flicker during a resizing operation than if it is left at its default **false** value.

When you override **Control.On*Xxx*( )** methods such as **OnPaint( )**, you should always have the first line in your method call **base.On*Xxx*( )** in order to assure that all the vdelegates attached to the event get called. After calling **base.OnPaint( ),** the first order of business is getting a reference to a **Graphics**. Instead of calling **Control.CreateGraphics( ),** an appropriate **Graphics** comes in as part of the **PaintEventArgs**. You do not have to worry about disposing of this **Graphics** at the end of the method (the Windows Forms infrastructure calls its **Dispose( )** method at the appropriate time).

To draw lines on a **Graphics,** you use an instance of the **Pen** class. **Pen**'s have various properties to control their appearance, but a **Pen** without a **Color** is meaningless, so you must specify a **Color** in the **Pen( )** constructor. (A shortcut for a simple pen of 1-pixel width with a predefined **Color** such as is used in this demo would be to use the **Pens** class: **Pens.Red** or **Pens.Blue**.)

The first time **OwnerDraw.OnPaint( )** is called, the **Pen** used is blue, subsequent paintings use a red one. The next several lines of **OnPaint( )** specify the sine wave: We're interested in drawing two sine wave cycles, and we want to draw the sine wave value at each pixel in the **Control**'s **Width**. So the **inc** variable holds the amount by which we'll count from 0 to 4π radians. The value returned from **Math.Sin( )** varies from -1 to 1. In order to fit these to the client area, the result is multipled by half the height and then half the height added to the result. This scales and transforms the values to fit in the client area (we'll talk about more efficient ways to do such steps later in the chapter).

We wish to draw a dot for each value we calculate, not a connected line. We accomplish this by specifying a **Rectangle** that is 1 unit in size at the calculated **x** and **y** coordinates.

The methods used for drawing betray how close GDI+ is to the underlying operating system. The **Graphics.Draw*Xxx*( )** methods are primitives, each one is implemented in some specialized, speed-optimized manner at the operating system level. This is also true of the **Graphics.Fill*Xxx*( )** methods that will be discussed shortly.

In this case, the drawing is done with a call to **Graphics.DrawRectangle( )** that takes the **Pen** and the **Rectangle** calculated previously. Once the rectangle is drawn, we increment the value of **x** and continue the loop.

The **SineWave( )** constructor first creates and places a blank **Panel** and a **Splitter** that are set to **DockStyle.Left**. The **OwnerDraw** is then set to **DockStyle.Fill**. When run, the **Panel p** will obscure the first part of the **OwnerDraw**'s client area: since **OwnerDraw** has no knowledge of the **Splitter**, the **OwnerDraw** actually fills the **SineWave**'s entire client area, **p** just obscures it. If you drag the **Splitter** to the left, you'll see more of the **OwnerDraw** come into view, but only the just-revealed portion will be drawn in red, as Windows will avoid repainting the still-exposed portion of the **OwnerDraw**.

Now, grab a corner of the **SineWave** application and resize it. On some computers, you'll see a flicker during redraw, but the console output will demonstrate that this is because **OnPaint( )** is constantly being called. You'll also see a large number of repaints if you take another window and drag it *over* the **SineWave** application.

Scaling and transforms

One thing that may have taken you aback when running **SineWave** is that the sine wave appears inverted – instead of starting at 0 and rising, it starts at 0 but moves towards the *bottom* of the **SineWave Form**. This is because Windows Forms default *coordinate system* is like that of a typewriter: **x** increases from right to left and **y** increases from the top to the bottom of the page:

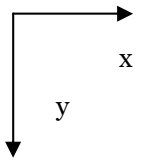


Figure 15-1: The default coordinate system of Windows Forms

If we wanted to have our sine wave appear so that positive is towards the top of the **Form** and negative towards the bottom, we could add the line

```
y *= -1;
```

to our calculations. Similarly, if instead of reaching all the way to the top and bottom, we wanted to consume only 90% of the space, we could use **y *= -0.9f** instead. If we wanted to combine this inversion and scaling with the transformation we need to make negative numbers appear, we could write:

```
y = -0.9f *(Height / 2 + Sin(d));
```

Naturally, we could do similar math with the **x** coordinate. Or we could use the **Graphics.ScaleTransform( )** to automatically do the multiplication for all values written to the context and **Graphics.TranslateTransform( )** to automatically add some value to all values written to the context.

This example uses these two methods to work directly with the values returned by **Math.Sin( )**:

```
//:c15:SineLine.cs
//Demonstrates Scaling and Transform
using System;
using System.Drawing;
using System.Windows.Forms;

class TransformPanel : Panel {
  internal TransformPanel(){
    ResizeRedraw = true;
  }

  protected override void OnPaint(PaintEventArgs e){
    base.OnPaint(e);
    Graphics g = e.Graphics;
    g.Clear(Color.White);
    Pen pen = new Pen(Color.Red);

    float widthScale = (float)
      (Width / (Math.PI * 4));
    float heightScale = Height / 2;
    float invertHeightScale = -heightScale;
    invertHeightScale *= .9f;
    Console.WriteLine("scale {0} {1}",
      widthScale, invertHeightScale);

    pen.Width = 1 / widthScale;

    //Set transforms for Graphics
    g.TranslateTransform(0, Height / 2);
    g.ScaleTransform(widthScale, invertHeightScale);

    PointF lastPoint = new PointF(0f, 0f);
    double inc = Math.PI * 4 / Width;
    for (float f = 0; f < Math.PI * 4; f += .1f) {
```

```
      float sin = (float) Math.Sin(f);
      PointF newPoint = new PointF(f, sin);
      g.DrawLine(pen, lastPoint, newPoint);
      lastPoint = newPoint;
    }
  }
}

class SineLine : Form {
  SineLine(){
    TransformPanel tPanel = new TransformPanel();
    tPanel.Dock = DockStyle.Fill;
    Controls.Add(tPanel);
  }

  public static void Main(){
    Application.Run(new SineLine());
  }
}///:~
```

Since we know that we're interested in drawing two cycles (4π radians) of the sine wave, we know that the resolution of our graph is **Width** / 4π. We calculate this value as **widthScale** in the **OnPaint( )** method of our **TransformPanel**. Similarly, we know that since the sine values range from -1 to 1, multiplying those values by ½ the **Height** will end up consuming the entire vertical space of the **Control**. This is the **heightScale** value that's calculated; **invertHeightScale** is the negative of that value (as we want positive numbers to be closer to the top of the **Form**). Finally, we multiply the **invertHeightScale** by **.9**, so that instead of taking up the entire vertical height, the results will consume 90% of the height.

The **Pen.Width** of the pen we're using begins with a default value of 1. **Graphics.TransformScale( )** works on *everything* in the context, though, so a **Pen** with **Width** = 1 will draw a line **Width** / 4π pixels wide! Therefore, we set **Pen.Width = 1 / widthScale**, which brings it back to being one pixel.

The line

```
g.TranslateTransform(0, Height / 2);
```

tells the **Graphics** to add nothing to all **x** values passed in and to add half the height to all the **y** values passed in, i.e., put the **y** axis halfway down the form.

The line

```
g.ScaleTransform(widthScale, invertHeightScale);
```

multiplies all **x** values by **widthScale** and all **y** values by **invertHeightScale**.

Transforms applied to the **Graphics** are cumulative (but obviously do not persist between one call to **OnPaint( )** and the next, as every time you are dealing with a new **Graphics** object). You can reset to the default, no-rotation, no-translation, transform (the *identity transform*) by calling **Graphics.ResetTransform( )**. Although transforms are cumulative, they are generally order-dependent (that is, translating and then rotating will have a different effect than rotating then translating). The mathematics of transforms will be covered in more detail a bit later.

Now that we're dealing with a scaled **Graphics**, we can no longer use integers to specify **Point**s on the canvas. The **Point(1, 1)** is at the top and more than 1/12th of the way across the form. Instead, we switch to the **PointF** structure, which allows us to specify locations in floating point.

Before entering our sine-calculating loop, we initialize **Point lastPoint** to the origin. Then, our loop increments **f** from 0 to 4 π in increments of 1/10th. The sine of **f** is calculated and **f** and **sin** are used directly to initialize a **PointF** value. If you stretch the original **SineWave** example, it breaks up into individual values; **SineLine** uses **Graphics.DrawLine( )** to connect the individual values as they're calculated.

It may seem to you that **SineLine** is not superior to **SineWave**, which may be true, but this example shows how transforms can dramatically reduce code length:

```
//:c15:SpinTheBottle.cs
//Demonstrates rotation transforms

using System;
using System.Drawing;
using System.Windows.Forms;

class BottleSpinner : Panel {
  internal BottleSpinner(){
    ResizeRedraw = true;
  }

  PointF[] pointer = new PointF[]{
    new PointF(0, 0), new PointF(.1f, .05f),
```

```
    new PointF(.09f, .2f), new PointF(.1f, .5f),
    new PointF(.02f, 1f),  new PointF(-.02f, 1f),
    new PointF(-.1f, .5f), new PointF(-.09f, .2f),
    new PointF(-.1f, .05f), new PointF(0, 0),
  };

  private float scale = .5f;
  public int PointerScale{
    get { return(int) (scale * 100);}
    set { scale = (float) value / 100;}
  }

  private int rot = 90;
  public int PointerRotation{
    get { return rot;}
    set { rot = value;}
  }

  protected override void OnPaint(PaintEventArgs ea){
    base.OnPaint(ea);
    Graphics g = ea.Graphics;
    g.Clear(Color.White);

    //Origin Offset = center of client area, plus some
    int xOrigin = Width / 2 + 50;
    int yOrigin = Height / 2;

    g.TranslateTransform(xOrigin, yOrigin);
    g.RotateTransform(rot);
    int smallerAxis = Width;
    if (Height < smallerAxis) {
      smallerAxis = Height;
    }
    float scaleTransform = smallerAxis  / 2 * scale;
    g.ScaleTransform(scaleTransform, scaleTransform);

    //Draw bottle
    Pen p = new Pen(Color.Red);
    p.Width = 1 / scaleTransform;

    g.DrawCurve(p, pointer);
```

```
  }
}

class SpinTheBottle: Form {
  BottleSpinner bs;
  TrackBar scaler;
  TrackBar spinner;

  SpinTheBottle(){
    Panel control = new Panel();
    control.Dock = DockStyle.Left;
    control.Width = 100;
    Splitter s = new Splitter();
    Controls.Add(s);
    Controls.Add(control);

    //Initializes controllers
    scaler = new TrackBar();
    scaler.Minimum = 1;
    scaler.Maximum = 100;
    scaler.Value = 50;
    scaler.TickFrequency = 10;
    scaler.Location = new Point(10, 10);
    scaler.Text = "Scale";
    scaler.ValueChanged +=
      new EventHandler(OnScaleChange);
    control.Controls.Add(scaler);

    spinner = new TrackBar();
    spinner.Minimum = 0;
    spinner.Maximum = 360;
    spinner.Value = 90;
    spinner.TickFrequency = 15;
    spinner.Location = new Point(10, 60);
    spinner.ValueChanged +=
      new EventHandler(OnSpinChange);
    control.Controls.Add(spinner);

    bs = new BottleSpinner();
    bs.Dock = DockStyle.Fill;
    Controls.Add(bs);
```

```
  }

  public void OnScaleChange(object src, EventArgs a){
    int scale = scaler.Value;
    bs.PointerScale = scale;
    bs.Invalidate();
  }

  public void OnSpinChange(object src, EventArgs a){
    int angle = spinner.Value;
    bs.PointerRotation = angle;
    bs.Invalidate();
  }

  public static void Main(){
    Application.Run(new SpinTheBottle());
  }
}///:~
```

The **BottleSpinner** panel first defines an array of **PointF**s that define (very roughly) a bottle shape using values from 0 to 1. Two properties, **PointerScale** and **PointerRotation,** will specify the transformation to be applied to the bottle shape.

OnPaint( ) calls **base.OnPaint( )** (as should always be done)**,** clears the canvas, and calculates the desired offset of the origin halfway across and down the **BottleSpinner**. Three transforms are then applied: **TranslateTransform( )** sets the origin, **ScaleTransform( )** sets the y-axis to increase towards the top of the screen, and **RotateTransform( )** sets the rotation equal to the value of the **rot** variable. Surprisingly, **RotateTransform( )** takes an angle in *degrees*, not radians.

After these transforms are applied, the real scaling transform is calculated from the value of the **scale** variable and the size of the **BottleSpinner** panel. **ScaleTransform( )** is called again; since transforms are additive, this scaling transform works with the previous **ScaleTransform( )** that flipped the y axis. A new **Pen** is created and its width set à la **DemoLine**.

Finally, **Graphics.DrawCurve( )** is used to draw the shape. **DrawCurve( )** draws *cardinal splines***,** a smooth curve that passes through all the points in the passed-in array.

The **SpinTheBottle Form** contains both a **BottleSpinner** panel and two **TrackBar** controls. **TrackBar**s, also called "slider" controls, are used to manipulate an integer value in a specified range. In this case, we specify that the **scaler** can have a range of 1 to 100 and the **spinner** a range of 0 to 360. Both have **ValueChanged** delegates that set the corresponding property in the **BottleSpinner bs** and then call **Invalidate( )**, which triggers the **OnPaint( )** event of our custom control.

Filling regions

So far, we have just used lines to draw on our **Graphics** context. Generally, in addition to (or in place of) drawing an outline, you'll want to fill a region. Just as lines are drawn with a series of **Draw*Xxx*( )** methods in the **Graphics** class, fills are drawn with a series of **Fill*Xxx*( )** methods. However, instead of using a **Pen** as the drawing tool, the **Fill*Xxx*( )** methods use a **Brush**. This example contrasts drawing and filling:

```
//:c15:RegionFill.cs
//Demonstrates filling a region
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RegionFill : Form {

  protected override void OnPaint(PaintEventArgs e){
    base.OnPaint(e);
    Graphics g = e.Graphics;
    ClientSize = new Size(120, 120);

    BackColor = Color.White;

    g.FillRectangle(Brushes.Red, 10, 10, 100, 100);
    g.DrawRectangle(Pens.Green, 10, 10, 100, 100);

    Pen pointer = new Pen(Color.Black);
    pointer.EndCap = LineCap.ArrowAnchor;
    g.DrawLine(pointer, 100, 7, 109, 7);

    g.DrawLine(pointer, 120, 7, 111, 7);
  }
```

```
  public static void Main(){
    Application.Run(new RegionFill());
  }
}///:~
```

Unlike previous examples, the owner-drawn **Control** in **RegionFill** is descended from **Form,** not **Panel**. This allows the sample programs to be slightly shorter, at the cost of losing any claim to decent object design. The **DrawRectangle** call uses a **Pen** from the **Pens** class and **FillRectangle** uses a **Brush** from the corresponding **Brushes** class.

The **Pen pointer** uses the **LineCap** enumeration that is part of the **System.Drawing.Drawing2D** namespace to add an arrow to lines drawn with the **Pointer**. Two lines are drawn to bracket the 110th pixel in the **Form**. When you run **RegionFill**, you'll see that the green rectangle is not entirely covered by the red fill even though both are given the same extents; edges of the green rectangle are still visible. The lines drawn by the **pointer** indicate that the **Draw*Xxx*( )** methods draw the *boundary* specified (in this case, [{10, 10}, {110, 110}]), while the **Fill*Xxx*( )** methods draw the interior (what is filled is [{9, 9}, {109, 109}]).

Although the **Pens** and **Brushes** classes are convenient, they do not expose an important feature of GDI+'s color model. The **Color** structure encapsulates a 32-bit color representation that includes an 8-bit transparent component (also known as an *alpha channel*) in addition to 8-bit components for each of the **Red**, **Green**, and **Blue** components.[1] An alpha value of 255 corresponds to a totally opaque color, while a value of 0 is totally transparent. In this example, we create **alphaGreen**, a somewhat transparent green, create a new **Brush** of that color, and overlay a rectangle filled with **alphaGreen** on a rectangle with **Color.Red**.

```
//:c15:AlphaFill.cs
//Demonstrates transparent color
using System;
using System.Drawing;
using System.Windows.Forms;
```

[1] While there are many color models, RGB is the dominant one for computer graphics, as it corresponds to the display components in monitors. The **Color** structure has methods to convert between RGB and Hue-Saturation-Brightness, a color model more popular with graphics designers.

```
class AlphaFill : Form {
  protected override void OnPaint(PaintEventArgs e){
    base.OnPaint(e);
    Graphics g = e.Graphics;
    ClientSize = new Size(130, 130);

    BackColor = Color.White;
    g.FillRectangle(Brushes.Red, 10, 10, 100, 100);

    Color alphaGreen =
    Color.FromArgb(227, 0, 255, 0);
    Brush aBrush = new SolidBrush(alphaGreen);
    g.FillRectangle(aBrush, 20, 20, 100, 100);
  }

  public static void Main(){
    Application.Run(new AlphaFill());
  }
}///:~
```

The line where the **Brush** is instantiated contains an upcast from **SolidBrush** to **Brush**. The next example illustrates all but one of the other subtypes of the abstract **Brush** class:

```
//:c15:BrushFill.cs
//Demonstrates filling a region
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class BrushFill : Form {

  protected override void OnPaint(PaintEventArgs e){
    base.OnPaint(e);
    Graphics g = e.Graphics;
    ClientSize = new Size(250, 250);
    BackColor = Color.White;

    Image img = Image.FromFile("images.jpg");
    Brush tBrush = new TextureBrush(img);
    g.FillRectangle(tBrush, 10, 10, 100, 100);
```

```
    Brush hBrush =
      new HatchBrush(HatchStyle.DiagonalCross,
        Color.Black, Color.White);
    g.FillRectangle(hBrush, 30, 90, 100, 100);

    Point startGradient = new Point(10, 120);
    Point endGradient = new Point(110, 220);
    Brush lBrush =
      new LinearGradientBrush(
        startGradient, endGradient,
        Color.Cyan, Color.Magenta);
    g.FillRectangle(lBrush, 10, 120, 100, 100);
  }

  public static void Main(){
    Application.Run(new BrushFill());
  }
}///:~
```

A **TextureBrush** tiles a region with the given **Image**, in this case one loaded from a file. A **HatchBrush** can draw various types of hatching, specified with the **HatchStyle** enumeration of the **Drawing2D** namespace; the hatch is drawn with the **Color**s specified in the **HatchBrush**'s **Foreground** and **Background** properties.

The **LinearGradientBrush** creates a smooth blend from one **Color** to another, from one **Point** to another. A **LinearGradientBrush** has a large number of properties to fine-tune the way the gradient is constructed. The **LinearGradientBrush** constructs a logical gradient between two **Point**s. These **Point**s need not be within the actual region being filled.

The only type of **Brush** not yet discussed is the **PathGradientBrush** which, like the **LinearGradientBrush**, is used to fill a region with a smooth blend of two colors. However, while the **LinearGradientBrush** creates a blend based on two logical **Point**s, the **PathGradientBrush** creates a blend based on the center and boundaries of a **GraphicsPath**.

A **GraphicsPath** is a series of connected lines and curves. The **GraphicsPath** used by the **PathGradientBrush** is considered to be closed (the last point on the path is considered connected to the first point on the path), so even if you create a path from just two lines, for the purposes of the gradient, the path will be considered a triangle. Like **LinearGradientBrush**, **PathGradientBrush** has

a wide variety of properties that can fine-tune the creation of the gradient, but this example demonstrates a basic **GraphicsPath** and a basic **PathGradientBrush**:

```
//:c15:PathGradientDemo.cs
//Demonstrates GraphicsPath and gradient fill
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class PathGradientDemo : Form {
  protected override void OnPaint(PaintEventArgs ea){
    base.OnPaint(ea);
    Graphics g = ea.Graphics;

    GraphicsPath path = new GraphicsPath();
    path.AddLine(0, 0, 250, 0);
    path.AddLine(250, 0, 0, 250);

    PathGradientBrush pgb =
      new PathGradientBrush(path);
    pgb.CenterColor = Color.Khaki;
    pgb.SurroundColors = new Color[]{
      Color.Red, Color.Green, Color.Blue};
    g.FillRectangle(pgb, 0, 0, 250, 250);
  }

  static void Main(){
    Application.Run(new PathGradientDemo());
  }
}///:~
```

The **GraphicsPath path** is a right triangle with legs of length 250; one leg and the hypotenuse are added explicitly, while we count on the **PathGradientBrush** to implicitly derive the third edge. We specify that we want a gradient with a khaki center. The **PathGradientBrush.SurroundColors** property specifies an array of colors corresponding to the endpoints of the components of the **GraphicsPath**. The color at any given point is a blend between the **CenterColor** and the two **SurroundColors** corresponding to the nearest points in the **GraphicsPath**.

Finally, to show the gradient, we use **FillRectangle( )**. Although the fill is for a rectangle, the **GraphicsPath** is triangular, so the appearance of the gradient is a triangle with a khaki center and red, green, and blue vertices.

Non-rectangular windows

Many multimedia applications have customizable interfaces ("skins") that prominently feature non-rectangular shapes. Programming this type of interface has traditionally required some pretty hard-core low-level stuff, but Windows Forms and GDI+ combine to make customized control shapes very simple. Each **Control** has a **Region** property that can be set to a **Region** containing a **GraphicsPath**. The **GraphicsPath** determines the shape of the **Control**. Since a **Form** is itself a **Control**, this can be used to create custom-shaped application windows.

```
//:c15:BinocularForm.cs
//Creates a non-rectangular application window
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;

class BinocularForm : Form {
  BinocularForm(){
    GraphicsPath gp = new GraphicsPath();
    gp.AddEllipse(0, 0, 200, 200);
    gp.AddEllipse(180, 00, 200, 200);
    this.Region = new Region(gp);
  }
  public static void Main(){
    Application.Run(new BinocularForm());
  }
}///:~
```

This is certainly the weirdest-looking example in this book and is definitely worth compiling and running. Notice that you can continue to resize the form by drag-clicking the border visible inside the right-hand ellipse. Note also that when you click in the hole created by the intersection of the two ellipses, the event passes "through" your application and activates the application you're running on top of. In order to create a "skinned" application, you would create a resource file (perhaps in XML) describing the graphics paths of all the customizable controls and their back- and foreground-colors, fonts, and so forth. To change the skin,

you'd simply create new **GraphicsPath**s and assign them to the appropriate controls.

Matrix transforms

GraphicsPath objects can be transformed, independently of the **Graphics** transforms by using the **Matrix** class. To understand the **Matrix** transforms, you must understand a small amount of matrix math.

An *affine transformation* is a rotation around the origin followed by a translation and is represented in matrix notation as:

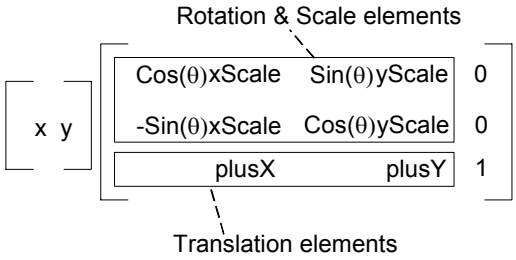


Figure 15-2: The elements of an affine transformation

This transformation would be expressed in this code:

```
newX = Math.Cos(theta) * xScale * x
     - Math.Sin(theta) * xScale * x
     + plusX;
newY = Math.Sin(theta) * yScale * y
     + Math.Cos(theta) * yScale * y
     + plusY;
```

The final column in an affine matrix is always the same. You can see how the **newX** value is derived by multiplying the first column of the 2-by-1 matrix (i.e., **x**) by each of the values in the first column of the 3-by-3 matrix, and then summing those results. Similarly, **newY** is derived by summing the products of the second columns.

Mostly, you will have no reason to calculate the **Matrix** elements directly. Instead, you'll start with the *identity transform*:

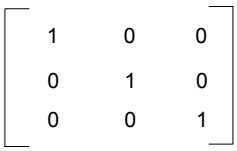


Figure 15-3: The identity transform

If you put these values into the above code, you'll see that the result is unrotated, unscaled, and untranslated. Then, you will call methods such as **Matrix.Rotate( )**, **Matrix.Scale( )**, and **Matrix.Translate( )** to calculate the new **Matrix** values.

A transform **Matrix** can be assigned to a **Graphics.Transform** property or passed as an argument to **GraphicsPath.Transform( )**. This example shows a custom control that displays the elements of an affine **Matrix**, another that displays a rectangle transformed by the **Matrix**, and an example of how the **Matrix** elements can be set directly.

```
//:c15:MatrixElements.cs
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

//Displays a 3 x 3 Matrix
class MatrixPanel : Panel {
  GraphicsPath bracket;

  public MatrixPanel(){
    ResizeRedraw = true;

    bracket = new GraphicsPath();
    bracket.AddLine(.1f, 0, 0, 0);
    bracket.AddLine(0, 0, 0, 1f);
    bracket.AddLine(0, 1f,.1f, 1f);
  }

  private Matrix matrix;
  public Matrix Matrix{
    get { return matrix;}
    set { matrix = value; Invalidate();}
  }
```

```
protected override void OnPaint(PaintEventArgs e){
  Graphics g = e.Graphics;

  DrawBrackets(g);
  DrawMatrix(g);
}

float scale = 0.9f;
float offset = 0.04f;

private void DrawBrackets(Graphics g){
  g.ScaleTransform(scale * Width, scale * Height);
  Pen p = new Pen(Color.Red);
  p.Width = 1 / Width;

  GraphicsContainer gState = g.BeginContainer();
  g.RotateTransform(180);
  g.TranslateTransform(-1 - offset, -1 - offset);
  g.DrawPath(p, bracket);
  g.EndContainer(gState);
  g.BeginContainer();
  g.TranslateTransform(offset, offset);
  g.DrawPath(p, bracket);
  g.EndContainer(gState);
}

private void DrawMatrix(Graphics g){
  if (matrix != null) {
    Font f = new Font("Arial", .1f);
    float[] els = matrix.Elements;
    PointF drawPoint = new PointF(0.05f, 0.1f);
    string s = els[0].ToString("0.00");
    g.DrawString(s, f, Brushes.Black, drawPoint);
    s = els[1].ToString("0.00");
    drawPoint.X = 0.4f;
    g.DrawString(s, f, Brushes.Black, drawPoint);
    s = els[2].ToString("0.00");
    drawPoint.X = 0.05f;
    drawPoint.Y = 0.4f;
    g.DrawString(s, f, Brushes.Black, drawPoint);
```

```
      s = els[3].ToString("0.00");
      drawPoint.X = 0.4f;
      g.DrawString(s, f, Brushes.Black, drawPoint);
      s = els[4].ToString("0.00");
      drawPoint.X = 0.05f;
      drawPoint.Y = 0.7f;
      g.DrawString(s, f, Brushes.Black, drawPoint);
      s = els[5].ToString("0.00");
      drawPoint.X = 0.4f;
      g.DrawString(s, f, Brushes.Black, drawPoint);

      //Draw 3rd col of affine
      drawPoint.X = .7f;
      drawPoint.Y = .1f;
      g.DrawString(
        "0.00", f, Brushes.Black, drawPoint);
      drawPoint.Y = .4f;
      g.DrawString(
        "0.00", f, Brushes.Black, drawPoint);
      drawPoint.Y = .7f;
      g.DrawString(
        "1.00", f, Brushes.Black, drawPoint);
    }
  }
}///:~ (Continues with TransformDisplay.cs)
```

The **GraphicsPath bracket** defines the shape of the tall square brackets that are used to display a matrix. The **bracket** shape is initialized in the **MatrixPanel( )** constructor that also sets **ResizeRedraw** to **true.**

The **Matrix** property of the **MatrixPanel** is used to get and set the associated **Matrix**. If the **Matrix** is assigned, the display should update to reflect its values, so a call to **Invalidate( )** is placed in the **set** method.

MatrixPanel.OnPaint( ) calls **DrawBrackets( )** and then **DrawMatrix( )**. **DrawBrackets** scales the **Graphics** so that a value of 1.0 is 90% of the **Height** or **Width** of the **Panel**.

Graphics.BeginContainer( ) and **EndContainer( )** can be used during complex transformation sequences to save the current state of the **Graphics( )**. Here, for instance, we save the state in a **GraphicsContainer gState** after the scaling transform, but then rotate and translate the **Graphics** to draw the right-hand bracket (the shape is defined as being a bracket that opens to the right, so to

draw the closing bracket, we have to flip it and move it over to the right-hand side of the **Panel**). After we're done, though, instead of reversing the translations, we just call **Graphics.EndContainer( )** with the state we wish to restore as an argument. Then, we can draw the left-hand bracket with just two lines of code.

MatrixPanel.DrawMatrix( ) first has to create a **Font** that's small enough to display on the scaled **Panel**. Then, the **Matrix** elements are retrieved and displayed in their proper positions. String formatting is used to constrain the lengths of the displayed data to two decimal places.

```
//:c15:TransformDisplay.cs
//Renders a rectangle transformed by a Matrix
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TransformDisplay : Panel {
  Matrix matrix;
  internal Matrix Matrix{
    set{ matrix = value;}
    get{ return matrix;}
  }

  internal TransformDisplay(){
    ResizeRedraw = true;
    matrix = new Matrix(1, 0, 0, 1, 0, 0);
  }

  protected override void OnPaint(PaintEventArgs e){
    Graphics g = e.Graphics;
    g.Clear(Color.White);
    g.Transform = matrix;
    Rectangle r = new Rectangle(0, 0, 100, 100);
    g.DrawRectangle(Pens.Red, r);
  }
}///:~ (Continues with MatrixAndTransform.cs)
```

TransformDisplay is a simple owner-drawn **Panel** that has a **Matrix** property, and, in **OnPaint( )**, applies this **Matrix** to its **Graphics** before drawing a red **Rectangle** from 0, 0 to 100, 100.

```
//:c15:MatrixAndTransform.cs
```

```
//Compile with
/*
csc MatrixAndTransform.cs TransformDisplay.cs
    MatrixElements.cs
*/
//Displays various matrices and their transforms
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MatrixAndTransform : Panel {
  internal MatrixAndTransform(Matrix m){
    MatrixPanel mp = new MatrixPanel();
    mp.Matrix = m;
    mp.Dock = DockStyle.Left;
    mp.Width = 120;
    TransformDisplay td = new TransformDisplay();
    td.Matrix = m;
    td.Dock = DockStyle.Left;
    Controls.Add(td);
    Controls.Add(mp);
  }
}

class MatrixElements : Form {
  MatrixElements(){
    Matrix m = new Matrix(1f, 0, 0, 1, 0, 0);
    MatrixAndTransform mt1 =
      new MatrixAndTransform(m);
    mt1.Dock = DockStyle.Top;
    Controls.Add(mt1);

    Matrix m2 = (Matrix) m.Clone();
    m2.Rotate(45);
    MatrixAndTransform mt2 =
      new MatrixAndTransform(m2);
    mt2.Dock = DockStyle.Top;
    Controls.Add(mt2);

    Matrix m3 = (Matrix) m.Clone();
```

```
    m3.Scale(0.25f, .5f);
    m3.Rotate(30);
    m3.Translate(125, -70);
    MatrixAndTransform mt3 =
      new MatrixAndTransform(m3);
    mt3.Dock = DockStyle.Top;
    Controls.Add(mt3);

    float rot = (float) Math.PI / 6;
    float xScale = .25f;
    float yScale = .5f;
    float el1 = (float) (Math.Cos(rot) * xScale);
    float el2 = (float) (Math.Sin(rot) * yScale);
    float el3 = (float) (-Math.Sin(rot) * xScale);
    float el4 = (float) (Math.Cos(rot) * yScale);
    float el5 = 36f;
    float el6 = 0;
    Matrix m4 =
      new Matrix(el1, el2, el3, el4, el5, el6);
    MatrixAndTransform mt4 =
      new MatrixAndTransform(m4);
    mt4.Dock = DockStyle.Top;
    Controls.Add(mt4);
    Height = 440;
  }

  public static void Main(){
    MatrixElements me = new MatrixElements();
    Application.Run(me);
  }
}///:~
```

The third custom control of this program is **MatrixAndTransform**, a **Panel** that combines a **MatrixPanel** and a **TransformDisplay** and sets them both to have the same **Matrix**.

MatrixElements is a **Form** that contains several **MatrixAndTransform**s. The **MatrixAndTransform mt1** is given the identity matrix to display. **mt2** uses **Matrix.Rotate( )** to rotate 45 degrees around the origin before drawing.

mt3 shows how transformations can accumulate. First, **Matrix.Scale( )** is used to scale the *x* and *y* dimensions by different amounts. Second, the **Matrix** is

rotated 30 degrees. Finally, the resulting scaled and rotated matrix is translated 125 units along the *x* axis and -70 on the *y* axis. Remember that this translation occurs *after* scaling and rotating, so these values are added along scaled, rotated axes (as will be apparent when compared to **mt4**).

For our final **MatrixAndTransform,** we're going to calculate the matrix's elements directly. Here we see the inconsistency between the rotation transformation methods (**Graphics.RotateTransform( )** and **Matrix.Rotate( )**) that use degrees, and the trigonometric functions of the **Math** class (**Math.Sin( )** and **Math.Cos( )** are needed here) that use radians. While **mt3** was rotated 30 degrees, the equivalent is π / 6 radians. This value, and the **xScale** and **yScale** values, are used to calculate the first 4 elements of the **Matrix**. The 5th and 6th elements, which are the translation elements, are set directly. These values will be added *directly to the screen coordinates*: setting them to 36 and 0 will end up having the same effect as the **m3.Translate(125, -70)** translation.

Figure 15-4 shows **mt4** on top and **mt3** below. You can see how the rotation and scaling elements (the four elements in the upper-left corner of both matrices) are identical, while the translation elements (the first two elements in the lowest row) are set precisely in **mt4** and a little off in **mt3**.

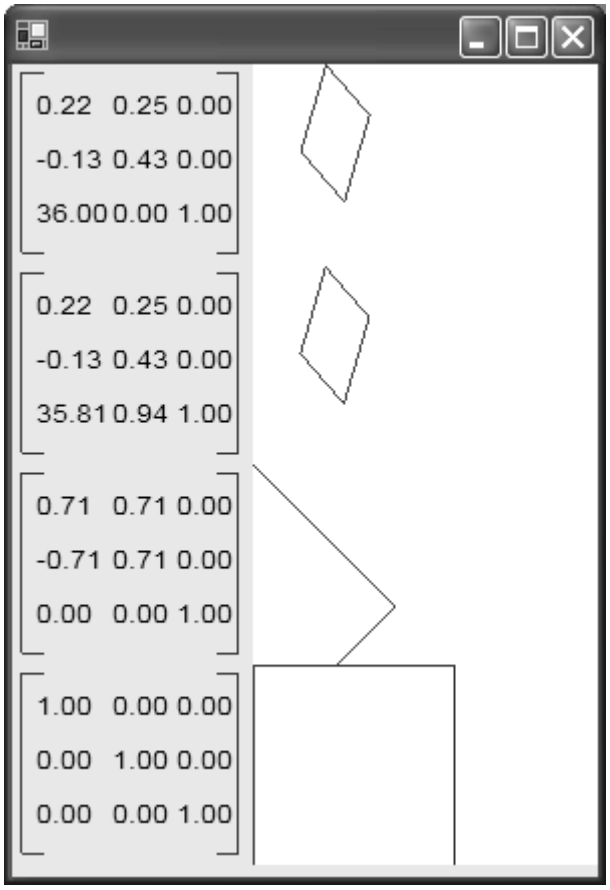


Figure 15-4: Various transforms and their effect

Trigonometry, matrix math, and linear algebra are the most important mathematical disciplines for programmers. They are of constant use, especially in game programming.

Hit detection

When programming GDI+, you'll generally want to react to mouse clicks near the shapes you are drawing. The **GraphicsPath** class has several methods to assist you. This example uses **GraphicsPath.IsVisible( )**, which returns **true** if a given point is within the **GraphicsPath**, to determine if the mouse was clicked within the desired shape:

```
//:c15:GraphicsPathHitTest.cs
//Demonstrates hit testing with a GraphicsPath
```

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class GraphicsPathHitTest : Form {
  GraphicsPath shape = new GraphicsPath();

  GraphicsPathHitTest(){
    shape.AddLine(10, 10, 30,10);
    Point[] curves = new Point[]{
      new Point(30, 10), new Point(70, 120),
      new Point(18, 240)
    };
    shape.AddCurve(curves);
    shape.AddLine(18, 240, 10, 10);
    shape.CloseFigure();

    this.MouseUp += new MouseEventHandler(HitTest);
  }

  protected override void OnPaint(PaintEventArgs ea){
    base.OnPaint(ea);
    Graphics g = ea.Graphics;

    g.DrawPath(Pens.Red, shape);
  }

  public void HitTest(object src, MouseEventArgs ea){
    Point mouseLocation = new Point(ea.X, ea.Y);
    if (shape.IsVisible(mouseLocation)) {
      Console.WriteLine("Clicked within path");
    }
  }

  public static void Main(){
    Application.Run(new GraphicsPathHitTest());
  }
}///:~
```

The **GraphicsPathHitTest( )** constructor builds a strange shape that combines straight lines and a cardinal spline. When done, **GraphicsPath.CloseFigure( )**

is used to connect the final point to the initial point (this is not necessary for this particular shape, which is already closed, but is a good habit to develop). After constructing the **shape,** an event handler is added to the **MouseUp** event.

The **OnPaint( )** method shows the **Graphics.DrawPath( )** method, which takes a **Pen** and a **GraphicsPath** (naturally, there is a **Graphics.FillPath( )** method as well). The **HitTest( )** method extracts the location of the mouse from the **MouseEventArgs** that are passed in, constructs a **Point** from them, and then uses **GraphicsPath.IsVisible( )** to determine if the mouse was clicked within the shape.

Fonts and text

Although the **RichTextBox** control can be used in many interfaces for creating a user interface featuring formatted text, the full power of Windows text support requires GDI+. We've already used the **Font** class as a property of a Windows Forms **Control** and in the **FontDialog** common dialog, but GDI+ allows a **Font** to be drawn with a **Brush** of any type. This example demonstrates that you can even draw text using a tiled image:

```
//:c15:FontDrawing.cs
//Basic text output in GDI+, using custom Brush
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class FontDrawing : Form {
  protected override void OnPaint(PaintEventArgs ea){
    base.OnPaint(ea);
    Graphics g = ea.Graphics;

    Font arial =
      new Font("Arial", 96, FontStyle.Regular);

    string hw = "Hello, C#";
    SizeF sizeOfString = g.MeasureString(hw, arial);
    if (Width < sizeOfString.Width + 20) {
      Width = (int) sizeOfString.Width;
    }

    Image img = Image.FromFile("images.jpg");
```

```
    TextureBrush tb = new TextureBrush(img);

    Point p = new Point(10, 10);
    g.DrawString(hw, arial, tb, p);
    Rectangle txtOrg = new Rectangle(10, 10, 2, 2);
    g.DrawEllipse(Pens.Red, txtOrg);
  }

  public static void Main(){
    Application.Run(new FontDrawing());
  }
}///:~
```

Since we want to be able to see at least some of the **Image** that we'll be drawing the **Font** with, we create a 96 point **Font**. Then, we use **Graphics.MeasureString( )** to determine the size that the **string** "Hello, C#" will require when drawn in the given context with the specified **Font**. If the current size of the **Form** is not big enough to accommodate the full text, the **Width** of the **Form** is increased (however, since **ResizeRedraw** is left at its default **false** value, the application window can be made smaller than the displayed text without triggering a repainting event).

A **TextureBrush** is created from a local image file, and **Graphics.DrawString( )** is called with the **string** to draw, the **Font** to use, the **Brush** to render the **Font** with, and the **Point** that corresponds to the upper-left corner of the rendered text. That point is circled in red; when you run this program, you may be surprised by how far from the text this appears.

If you find yourself using GDI+ to draw text on the screen, you probably are interested in drawing the text in strange ways (vertically, diagonally, etc.). You can use the various transformation methods in the **Graphics** class, as this example shows (before running this program, see if you can predict what the output will look like):

```
//:c15:TextTransform.cs
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TextTransform : Form {

  protected override void OnPaint(PaintEventArgs ea){
```

```
    base.OnPaint(ea);
    Graphics g = ea.Graphics;

    Font arial =
      new Font("Arial", 24, FontStyle.Regular);

    g.RotateTransform(90);
    g.ScaleTransform(1.5f, -1);
    string hw = "Hello, C#";

    Rectangle r = new Rectangle(10, 10, 100, 100);
    g.DrawString(hw, arial, Brushes.Black, r);
    g.DrawRectangle(Pens.Red, r);
  }

  public static void Main(){
    Application.Run(new TextTransform());
  }
}///:~
```

In addition to rotating and scaling the drawing canvas, a **Rectangle**, not a **Point**, is used as the final argument to **Graphics.DrawString( )**. This overload of **DrawString( )** wraps and clips the text to the **Rectangle**.

Printing

Now that you've had a whirlwind tour of GDI+, you can finally print from Windows Forms. Printing is done with a **PrintDocument**, an object with a **PrintPage** event. You attach a delegate to this event and receive an instance of **PrintPageEventArgs**, which includes a **Graphics**. This **Graphics** object corresponds to one page of your output device. You draw graphics on the **Graphics**, just as you would in an **OnPaint( )** method. When done, the page will be printed.

In this example, we define a simple form that calls both the **PrintPreviewDialog** and **PrintDialog** common dialogs:

```
//:c15:Printing.cs
//Demonstrates printing from Windows Forms
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;
```

```
class Printing : Form {
  PaperWaster pw = new PaperWaster();
  Printing(){
    MainMenu menu = new MainMenu();
    Menu = menu;

    MenuItem fMenu = new MenuItem("&File");
    menu.MenuItems.Add(fMenu);
    MenuItem prvMenu = new MenuItem("P&review...");
    fMenu.MenuItems.Add(prvMenu);
    prvMenu.Click += new EventHandler(OnPreview);
    MenuItem prtMenu = new MenuItem("&Print...");
    fMenu.MenuItems.Add(prtMenu);
    prtMenu.Click += new EventHandler(OnPrint);
  }

  public void OnPreview(object src, EventArgs ea){
    PrintPreviewDialog ppd =
      new PrintPreviewDialog();
    ppd.Document = pw.Document;
    if (ppd.ShowDialog() == DialogResult.OK) {
      //Dialog showed okay
    }
  }

  public void OnPrint(object src, EventArgs ea){
    PrintDialog pd = new PrintDialog();
    pd.Document = pw.Document;
    if (pd.ShowDialog() == DialogResult.OK) {
      //Dialog showed okay
    }
  }

  public static void Main(){
    try{
      Application.Run(new Printing());
    }catch(Exception ex){
      Console.WriteLine(ex);
    }
  }
```

```
}

class PaperWaster {
  PrintDocument pd = new PrintDocument();
  internal PrintDocument Document{
    get { return pd;}
  }

  internal PaperWaster(){
    pd.PrintPage +=
      new PrintPageEventHandler(PrintAPage);
  }

  void PrintAPage(object src, PrintPageEventArgs ea){
    Graphics g = ea.Graphics;
    Font f = new Font("Arial", 36);
    SolidBrush b = new SolidBrush(Color.Black);
    PointF p = new PointF(10.0f, 10.0f);
    g.DrawString("Reduce, Reuse, Recycle", f, b, p);
    ea.HasMorePages = false;
  }
}///:~
```

Both the **PrintPreview** and **PrintDialog** classes have a **Document** property which must be set to an instance of class **PrintDocument**. This is done in the **OnPreview( )** and **OnPrint( )** event handlers in the **Printing** form. Our domain class **PaperWaster** has a **Document** property that returns a **PrintDocument** whose **PrintPage** event has been delegated to **PaperWaster.PrintAPage( )**.

The only part of **PrintAPage( )** that is new is setting the **HasMorePages** property of the **PrintPageEventArgs** argument to **false**. This is actually unnecessary, as **false** is its default value, but if set to **true**, **PrintAPage** will be called again. It is up to you to maintain the state of your domain object so that a sequence of calls to **PrintAPage( )** properly output pages in order and then terminate by setting **HasMorePages** to **false.**

Bitmaps

Since we've already shown the use of an **Image** inside of a **TextureBrush**, it shouldn't be a shock that GDI+ supports displaying **Image**s directly. To display an **Image** you already have in memory, you use **Graphics.DrawImage( )**,

which has a variety of overloads that allows you to display the image or a portion of it in original size or scaled and as a parallelogram. This example shows some of these overloads:

```
//:c15:BitMapDemo.cs
//Demonstrates basic bitmap manipulation
using System;
using System.Drawing;
using System.Windows.Forms;

class BitMapDemo : Form {
  Bitmap bmp;

  BitMapDemo(){
    bmp = new Bitmap("tumbuan.jpg");
  }
  protected override void OnPaint(PaintEventArgs ea){
    base.OnPaint(ea);
    Graphics g = ea.Graphics;
    g.DrawImage(bmp, new Point(70, 70));

    Rectangle scaled = new Rectangle(20, 20, 60, 60);
    g.DrawImage(bmp, scaled);
    g.DrawRectangle(Pens.Red, scaled);

    Point[] pGram = new Point[]{
      new Point(50, 10),
      new Point(100, 30),
      new Point(20, 100),
    };
    g.DrawImage(bmp, pGram);
  }

  public static void Main(){
    Application.Run(new BitMapDemo());
  }
}///:~
```

The demo loads a **Bitmap** that is included in the book's source code file. The first **DrawImage( )** call draws the **Image**, unscaled, at the specified **Point**. The second **DrawImage( )** scales the bitmap to fit in the **Rectangle**.

The third overload of **DrawImage( )** accepts an array of 3 **Point**s; these **Point**s define a parallelogram. The first **Point** is the origin, the second point is the upper-right corner of the parallelogram. The **Image** will be drawn to follow the slope defined by these two points. The third **Point** defines the lower-left corner of the parallelogram and the fourth corner of the parallelogram is inferred. If you pass an incorrectly sized array to this method, **DrawImage( )** will throw an exception.

The easiest way to draw on an **Image** is to use the static method **Graphics.FromImage( )**, which returns a **Graphics** on which you can use the gamut of GDI+ drawing tools. This example loads a bitmap, draws on it, and saves the result to disk.

```
//:c15:ImageDrawAndSave.cs
//Demonstrates how to draw on an Image
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageDrawAndSave : Form {
  Image img;
  PictureBox pb;
  string fName;

  ImageDrawAndSave(){
    MainMenu mm = new MainMenu();
    Menu = mm;

    MenuItem fMenu = new MenuItem("&File");
    mm.MenuItems.Add(fMenu);

    MenuItem oMenu = new MenuItem("&Open...");
    oMenu.Click += new EventHandler(OpenImage);
    fMenu.MenuItems.Add(oMenu);

    MenuItem sMenu = new MenuItem("&Save...");
    sMenu.Click += new EventHandler(SaveImage);
    fMenu.MenuItems.Add(sMenu);

    pb = new PictureBox();
    pb.Dock = DockStyle.Fill;
    Controls.Add(pb);
```

```
}

public void OpenImage(object src, EventArgs ea){
  OpenFileDialog ofd = new OpenFileDialog();
  ofd.Filter = "Image files (*.bmp;*.jpg;*.gif)"
    + "|*.bmp;*.jpg;*.gif;*.png";
  DialogResult fileChosen = ofd.ShowDialog();
  if (fileChosen == DialogResult.OK) {
    try {
      img = Image.FromFile(ofd.FileName);
      fName = ofd.FileName;
      pb.Image = img;
      DrawOnImage(img);
    } catch (Exception e) {
      Console.WriteLine(e);
    }
  }
}

void DrawOnImage(Image img){
  Graphics g = Graphics.FromImage(img);
  using(g){
    g.FillRectangle(Brushes.Red, 10, 10, 20, 20);
  }
  pb.Invalidate();
}

public void SaveImage(object src, EventArgs ea){
  if (img != null) {
    SaveFileDialog sfd = new SaveFileDialog();
    sfd.FileName= fName;
    DialogResult saveChosen = sfd.ShowDialog();
    if (saveChosen == DialogResult.OK) {
      img.Save(sfd.FileName);
    }
  }
}

public static void Main(){
  Application.Run(new ImageDrawAndSave());
}
```

```
}///:~
```

The program uses **OpenFileDialog** and **SaveFileDialog** as discussed in the previous chapter. After the **Image** is opened, **DrawOnImage( )** generates a **Graphics** for it and draws a red rectangle on the image. In order to ensure that the **Graphics** is disposed of properly, it's given as an argument to a **using** block. Once drawn upon, **pb.Invalidate( )** is called to trigger a repaint of the **PictureBox**.

Although using a **Graphics** allows the use of all of GDI+ tools, when an **Image** is a **Bitmap,** you can directly set and get the color of individual pixels using **Bitmap.GetPixel( )** and **Bitmap.GetPixel( )**. This example randomly speckles a bitmap with randomly colored pixels:

```
//:c15:ManipPixels.cs
//Shows direct pixel manipulation in a Bitmap
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;

class ManipPixels : Form {
  PictureBox pb;

  ManipPixels(){
    MainMenu mm = new MainMenu();
    Menu = mm;

    MenuItem fMenu = new MenuItem("&File");
    mm.MenuItems.Add(fMenu);

    MenuItem oMenu = new MenuItem("&Open...");
    oMenu.Click += new EventHandler(OpenImage);
    fMenu.MenuItems.Add(oMenu);

    pb = new PictureBox();
    pb.Dock = DockStyle.Fill;
    Controls.Add(pb);
  }

  public void OpenImage(object src, EventArgs ea){
    OpenFileDialog ofd = new OpenFileDialog();
```

```
    ofd.Filter = "Image files (*.bmp;*.jpg;*.gif)"
      + "|*.bmp;*.jpg;*.gif;*.png";
    DialogResult fileChosen = ofd.ShowDialog();
    if (fileChosen == DialogResult.OK) {
      try {
        Bitmap bmp = new Bitmap(ofd.FileName);
        pb.Image = bmp;
        RandPixels(bmp);
      } catch (Exception e) {
        Console.WriteLine(e);
      }
    }
  }

  void RandPixels(Bitmap bmp){
    Random rand = new Random();

    int imgSize = bmp.Width * bmp.Height;
    int iToChange = (int) (imgSize * .25);

    for (int i = 0; i < iToChange; i++) {
      int x = rand.Next(bmp.Width);
      int y = rand.Next(bmp.Height);

      int r = rand.Next(255);
      int g = rand.Next(255);
      int b = rand.Next(255);
      Color c = Color.FromArgb(255, r, g, b);

      bmp.SetPixel(x, y, c);
    }

    pb.Invalidate();
  }

  public static void Main(){
    Application.Run(new ManipPixels());
  }
}///:~
```

RandPixels calculates the total number of pixels in the **Bitmap** by multiplying the **Width** by the **Height**. The loop runs so that approximately one-fourth of the

pixels in the image are changed (only approximately because the same pixel may be chosen in the loop). A random coordinate (**x**, **y**) is chosen and a random **Color** created. **Bitmap.SetPixel( )** makes the change. When the loop is done, **pb.Invalidate( )** causes a repaint. **Bitmap.GetPixel( )** is similarly straightforward: given a coordinate, it returns a **Color**.

Rich clients with interop

One of the real joys of working with the .NET Framework after spending several years programming for browser-based interfaces is the rediscovery of the power of the client machine. Heck, just having normal menus again is a thrill. Windows Forms provides many powerful components, but there are many additional components available to Windows users. You can use these components in two ways:

- If the component supports Microsoft's Component Object Model (COM), you can generate a Runtime Callable Wrapper (RCW) proxy that allows you to use the component much as if it were a native .NET object.
- If the component is a .DLL that exports functions, you can write your own wrapper class that accesses the functions as if they were static methods.

There are two significant problems with working with Interop: documentation of the non-.NET component and the component's implementation quality. Sadly, neither of these can be taken for granted and there's little that can be done about it. .NET's threading and memory models are the best kind: sophisticated enough to be easy. Before using *any* component, .NET or otherwise, you should perform due diligence by searching newsgroup archives for pointers to bugs, resource issues, and programming quirks.

COM Interop and the WebBrowser control

If the **RichTextBox** doesn't provide the display capabilities you need, perhaps the core HTML component of Internet Explorer will be sufficient. Since Internet Explorer 4, Microsoft has made its browser's components available as COM components.

COM Interop is enabled by the generation of a *Runtime Callable Wrapper*, an assembly that mediates between the .NET world and the COM world. Tools provided in the .NET Framework SDK automatically can generate these wrappers from COM typelibs. The general solution to creating a wrapper is to use the tool

tlbimp. If the component you are attempting to import is an ActiveX control (COM's version of the visual **Component**s in Windows Forms), the **aximp** tool automatically creates a proxy for the ActiveX control that makes visual programming far easier, as it exposes .NET-style properties and events.

The Internet Explorer controls are available as an ActiveX control stored in the file **shdocvw.dll**, which you should have in your **\Windows\System32** directory. Assuming that this is the case and that you've installed Windows in drive C, you create a Runtime Callable Wrapper with this command line:

```
aximp c:\windows\system32\shdocvw.dll
```

This should generate two .NET assemblies: **SHDocVw.dll** and **AxSHDocVw.dll**. **AxSHDocVw.dll** contains the .NET proxy for the ActiveX control and relies on **SHDocVw.dll**. Your code only needs to *reference* **AxSHDocVw**, but both assemblies (and, of course, the original COM component) need to be available for your .NET program to run.

Programming an ActiveX with COM Interop requires just a little bit more attention to detail than programming in Windows Forms:

```
//:c15:COMInterop.cs
//Compile with:
//csc /reference:AxSHDocVw.dll COMInterop.cs
//Demonstrates COM Interop with the WebBrowser ctl
using System;
using System.Drawing;
using System.Windows.Forms;
using AxSHDocVw;

class COMInterop : Form {
  AxWebBrowser browserCtl;
  COMInterop() {
    browserCtl = new AxWebBrowser();
    browserCtl.BeginInit();
    browserCtl.Dock = DockStyle.Fill;
    browserCtl.NavigateComplete2 +=
      new
DWebBrowserEvents2_NavigateComplete2EventHandler(
  DoNavigationComplete);
    browserCtl.EndInit();
    Controls.Add(browserCtl);
```

```
    this.Load += new EventHandler(DoNavigate);
  }

  protected override void Dispose(bool b){
    if (b) {
      foreach(Control c in Controls){
        c.Dispose();
      }
    }
    base.Dispose(b);
  }

  void DoNavigate(object src, EventArgs ea) {
    string url = "http://www.ThinkingIn.Net/";
    object flags = 0;
    object tgtFrame = "";
    object httpPostData = "";
    object addlHeaders = "";

    browserCtl.Navigate(url, ref flags,
      ref tgtFrame, ref httpPostData,
      ref addlHeaders);
  }

  void DoNavigationComplete(object src,
    DWebBrowserEvents2_NavigateComplete2Event e){
    Console.WriteLine("Successfully surfed to: "
      + e.uRL);
  }

  [STAThread]
  public static void Main() {
    Application.Run(new COMInterop());
  }
}///:~
```

The COM wrapper defines the namespace **AxSHDocVw**, which contains a class called **AxWebBrowser**. Visual Studio .NET provides code completion and object browsing tools for COM objects; command-line users can use **ILDasm** and Web searches to get a handle on programming an unfamiliar COM component.

After instantiating an **AxWebBrowser** object in the **COMInterop( )** constructor, calls to manipulate the initial state of the ActiveX object are wrapped within calls to **BeginInit( )** and **EndInit( )**. These methods ensure that the ActiveX is not displayed before it is completely initialized; they may not be necessary if you are not going to use a design-time tool such as Visual Studio's Designer tool but when dealing with unmanaged code, safe is better than sorry. Similarly, when using Interop, always implement a **Dispose(bool)** method à la that shown here.

The **AxWebBrowser** initialization code sets the control to fill the **COMInterop** form's client area and attaches a delegate to the event associated with navigation completing.

During development, calls to **AxWebBrowser.Navigate( )** failed when used within the **COMInterop( )** constructor, so the **DoNavigate( )** method is attached to the **COMInterop**'s **Load** event. The **DoNavigate( )** method just sets up the arguments to the **browserCtl AxWebBrowser** object's **Navigate( )** method. The most important variable is, of course, the URL. Information on the other arguments (and a comprehensive reference to the WebBrowser control) can be found on MSDN.

The event handler **DoNavigationComplete( )** simply outputs the surfed-to URL when navigation is completed.

COM Interop requires that the executing thread's **ApartmentState** property be set to **ApartmentState.STA**, which stands for *single-threaded apartment model*. The easiest way to do this is to apply the **[STAThread]** attribute to the static **Main( )** method. You can ignore the **ApartmentState** property in all other circumstances.

To compile **COMInterop**, use the command-line:

```
csc /reference:AxSHDocVw.dll COMInterop.cs
```

Happy surfing!

COM Interop challenges

Okay, now that you've had a *good* experience with COM Interop, it's time for a little no-silver-bullet reality. Using COM Interop to work with existing Windows systems *may* go very well, but there *may* be significant challenges. Specifically, the biggest challenge emerging with COM Interop is that many COM systems are written in a way that is dependent on deterministic finalization (the COM component expects its cleanup-and-release code to be called explicitly), while

COM Interop relies on the garbage collector, which as discussed in chapter 5, is non-deterministic. Design and implementation of complex COM Interop scenarios is beyond the scope of this book, but suffice it to say that there are specific calls (such as **Marshal.ReleaseComObject( )**) that provide the needed functionality, but may also require significant amounts of design and implementation effort to work with a complex, legacy COM system.

Non-COM Interop

Probably most of your desire to interact with the unmanaged world will be via COM and ActiveX objects. However, it is also possible to use unmanaged "plain vanilla" DLLs from C#. Here, even more than with COM Interop, you have to take pains to properly initialize and dispose of the unmanaged objects.

To demonstrate non-COM Interop, we're going to access what's probably the most actively used piece of code in the world – the DLL that's responsible for drawing the cards that are used in Windows Solitaire. The card-drawing routines are stored in an unmanaged DLL called **cards.dll** that is found in the **\system32** subdirectory of your Windows directory.

The DLL has a function **cdtTerm( )** that should be called when your use of the DLL has ended. We can't just put a call to **cdtTerm( )** in a **Dispose( )** or a destructor, since those are called for every *instance* of the card created. Instead, we need to have a reference count to the number of cards that have been created and destroyed. When the reference count gets to 0, only then should we call **cdtTerm( )**.

The example demonstrates interoperability with non-COM components: the most general level of Windows interoperability. We used **dumpbin /exports c:\windows\system32\cards.dll** and some Internet research to discover the appropriate function signatures.

```
//:c15:CardPainter.cs
//Demonstrates non-COM Interop,
//and the ever-important ability to draw cards!
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;
using System.Runtime.InteropServices;
```

```
class CardHolder : Panel {
  [DllImport("cards.dll")]
  public static extern
  long cdtInit(ref long x, ref long y);
  [DllImport("cards.dll")]
  public static extern long cdtTerm();
  [DllImport("cards.dll")]
  public static extern
  long cdtDrawExt(IntPtr hdc, long x, long y,
    long dx, long dy, long ordCard,
    long iDraw, long color);
  [DllImport("cards.dll")]
  public static extern
  long cdtDraw(IntPtr hdc, long xOffset,
    long cardChosen, long dx, long dy, long color);
  [DllImport("cards.dll")]
  public static extern
  long cdtAnimate(IntPtr hdc, long ordCard,
    long x, long y, long animationType);

  static int dllRefCount = 0;
  static int defX, defY;

  void InitDLL(){
    long x = 0;
    long y = 0;
    Console.WriteLine("Initializing DLL");
    cdtInit(ref x,  ref y);
    defX = (int) x;
    defY = (int) y;
  }

  internal CardHolder(int iCard){
    cardChosen = iCard;
    lock(typeof(CardHolder)){
      Interlocked.Increment(ref dllRefCount);
      if (dllRefCount == 1) {
        InitDLL();
      }
    }
    Width = defX;
```

```
    Height = defY;
  }

  int cardChosen = 0;

  protected override void OnPaint(
    PaintEventArgs pea){
    Graphics g = pea.Graphics;
    pea.Graphics.Clear(Color.White);
    IntPtr handle = g.GetHdc();
    try {
      cdtDraw(
        handle, 0, cardChosen,
        this.Width, this.Height, 0);
    } finally {
      g.ReleaseHdc(handle);
    }
  }

  protected override void Dispose(bool b){
    lock(typeof(CardHolder)){
      Interlocked.Decrement(ref dllRefCount);
      Console.WriteLine(dllRefCount);
      if (dllRefCount == 0) {
        Console.WriteLine("Finalizing dll");
        cdtTerm();
      }
    }
  }
}

public class CardPainter : Form {
  ArrayList cards = new ArrayList();
  static Random r = new Random();
  int offset = 10;

  public CardPainter() {
    Button draw = new Button();
    draw.Text = "Deal";
    draw.Click += new EventHandler(OnDrawCard);
    draw.Location = new Point(10, 10);
```

```
    Controls.Add(draw);

    Button discard = new Button();
    discard.Text = "Discard";
    discard.Click += new EventHandler(OnDiscard);
    discard.Location = new Point(10, 60);
    Controls.Add(discard);
  }

  public void OnDrawCard(object src, EventArgs e){
    CardHolder ch = new CardHolder(r.Next(51));
    offset += 10;
    ch.Location = new Point(offset, offset);
    cards.Add(ch);
    Controls.Add(ch);
  }

  public void OnDiscard(object src, EventArgs e){
    if (cards.Count > 0) {
      CardHolder ch =
        (CardHolder) cards[cards.Count - 1];
      cards.Remove(ch);
      Controls.Remove(ch);
      offset -= 10;
      Invalidate();
    }
  }

  protected override void Dispose(bool b){
    if (b) {
      foreach(Control c in cards){
        c.Dispose();
      }
    }
    base.Dispose(b);
  }

  public static void Main(){
    Application.Run(new CardPainter());
  }
}///:~
```

The program references the **System.Threading** namespace because it uses the **Interlocked** class for reference counting; this example is single-threaded, but the **Interlocked** class provides thread-safe reference counting, which is generally very important when working with unmanaged code (we'll revisit **Interlocked** in the next chapter).

The program also references the **System.Runtime.InteropServices** namespace, which supplies the **DllImportAttribute** type. In the example, the **DllImportAttribute**s specify that we're importing functions from **cards.dll**, which is not a .NET assembly, but an old fashioned Windows dynamic link library. Since C-based DLLs do not have objects and methods, their functions are the equivalent of static methods. The **extern** keyword specifies that the implementation is external – located in the DLL specified in the **[DllImport]** attribute.

We're wrapping the calls to **cards.dll** in a custom control descended from **Panel**. The **CardHolder( )** constructor takes an integer, which should be in the range 0 to 51, that specifies the card.

The static variable **dllRefCount** contains the number of **CardHolder**'s that have been created. We want to call **cdtInit( )** when that number is 1, and **cdtTerm( )** when that number returns to 0. The **lock** keyword and the block that follows will be discussed in length in the next chapter; for the moment just know that the purpose here is to ensure that only one thread at a time can access the **dllRefCount** variable.

If **dllRefCount** is 1, the method **InitDLL( )** is called. In turn, it calls **cdtInit**, passing in two arguments that, subsequent to the call, contain the default width and height of the cards that the DLL will draw (on my machines, these seem to always be 71 and 96 – perhaps they vary with screen resolution). These values are stored in the static variables **defX** and **defY**; back in the **CardHolder( )** constructor, these variables are used to set the size of the **CardHolder** panel.

CardHolder.OnPaint( ) uses a method within the **cards.dll** to draw on the screen. As a relic of the by-gone days of Windows programming, **cdtDraw( )** takes as its first parameter a *device context handle* (an **hDC**, represented in .NET as an **IntPtr**). **hDC**'s must be requested and released explicitly with calls to **Graphics.GetHdc( )** and **Graphics.ReleaseHdc( )**. In order to be responsible, it's a good habit to put all usage of an **hDC** in a **try…finally** block.

The final argument to **cdtDraw( )** is an integer between 0 and 51, representing the various cards in the deck. This is the taken from the **cardChosen** variable that was passed to the **CardHolder( )** constructor.

The **CardHolder.Dispose( )** method is, of course, called for each object of type **CardHolder**. But we do not want to call **cdtTerm( )** until there are *no* objects of type **CardHolder** in the program; we also do not want to call it if the "Discard" button has been pressed erroneously. Thus we have to use **Interlocked.Decrement( )** to complement the **Interlocked.Increment( )** called in the constructor, and because we're pretending that **CardHolder** might be used in a multithreaded environment, we again **lock** the class's **Type** to ensure that **Interlocked.Decrement( )** cannot be called twice before the comparison of **dllRefCount** to 0.

The **CardPainter** class puts two **Button**s on the form and attaches the **OnDrawCard( )** and **OnDiscard( )** methods to their **Click** events. **OnDrawCard( )** instantiates a new **CardHolder** for a random card and places it on the **CardPainter** client area, incrementing the **offset** instance variable so that subsequent cards will appear tiled. The **CardHolder** is also added to an **ArrayList cards**.

OnDiscard( ) uses the **cards** collection as an easy way to track the various **CardHolder**s. If there are any **CardHolder**s in **cards**, the last **CardHolder** is retrieved and removed from the **cards** collection and from the form.[2]

As always when using either COM or non-COM Interop, we implement a **Dispose( )** method that calls the **Dispose( )** method of our interoperating classes. When the **CardPainter** application is closed, you will see **CardHolder.Dispose( )** count down the **CardHolder.dllRefCount** variable and, when it reaches 0, the call to **cdtTerm.** If the "Discard" button is pressed more times than the "Deal" button, you'll still see that **cdtTerm** is called only when the **dllRefCount** goes to 0.

Summary

GDI+ provides a stateless drawing context that can be used to fully customize your user interface. Access to GDI+'s capabilities come by way of the **Graphics** class, which is a low-level "canvas" on which you can draw lines and curves, text,

---

[2] The **CardPainter** class is not thread-safe. We kept our "forward referencing" to threading issues limited to the **CardHolder** class.

images, and complex shapes. Windows underlying repainting algorithms, which are designed to reduce flickering in the user interface, can be somewhat confusing, only redraw the portion of the display that has been marked "dirty" by the underlying Windows logic. This mostly causes confusion while redrawing a window that is being resized; setting **Control.ResizeRedraw** to **true** is usually sufficient for eliminating this problem.

Instances of type **Graphics** are short-lived and associated with Windows internal data-structures that cannot be recovered by garbage collection. Therefore, you must dispose of **Graphics** either by explicitly calling their **Dispose( )** method within a **finally** block or by using the **Graphics** as an argument to a **using** block. **Graphics** can be transformed in a variety of ways, including scaling, translating (moving), rotating, and shearing transforms.

Graphics contains a series of primitive methods which draw lines and fill regions; the characteristics of a line being drawn are encapsulated in a **Pen** object, the characteristics of a region being filled are within a **Brush**. **Graphics** also has primitives for displaying text and images. The characteristics of the text being drawn are encapsulated within a **Font**, and **Graphics.MeasureString( )** is an important method for determining the size required to display some given text. If **Graphics.DrawString( )** is passed a **Rectangle** instead of a **Point**, it will wrap the text to the width of the **Rectangle**, if the wrapped text cannot fit in the **Rectangle**, the text will be clipped.

Images can be drawn on **Graphics** in their original size, scaled, or as a parallelogram; in combination with the transformations that can be performed on the **Graphics** itself, and on **GraphicsPath**s, essentially any graphical effect can be achieved.

Exercises

1. Write a custom control that draws graduated diagonal lines from corner to corner. Place this custom control in a standard Windows Form and confirm that it docks and lays itself out appropriately.

2. Refactor the previous exercise so that either inheritance or a delegate is used to draw the graduated lines so that they scale (hashmarks at 1/8th-length positions) or maintain an absolute size (hashmarks every 1/8th of an inch).

3. Write a program that turns mouse-clicks into points in a curve and allows you to select the color and pixel-width of the pen that draws the curve.

4. Add the ability to place text, geometric shapes, and arrows to the program developed in Exercise 3.

5. Refactor the previous example so that the shape closest to the mouse point becomes editable (allow deletion and movement).

6. Add the ability to fill shapes to the shape-drawing program.

7. Refactor the previous shape-drawing program so that it defines a shape-drawing interface and can dynamically load assemblies implementing that interface. Implement the interface with a class that draws a spiral.

8. (Advanced) Using the **System.Reflection.Emit** namespace (not discussed in this book) and others, write a program that dynamically compiles and graphs a function typed into a text box.

9. Add the ability to scale and rotate shapes to the shape-drawing program.

10. Add the ability to print to your shape-drawing program.

11. Using everything you have learned, write a solitaire game.

12. Using the past two chapters and the wavelet transform code from Chapter 10, write a wavelet explorer program that allows you to view, transform, and manipulate wavelet-transformed graphical images.

16: Multithreaded Programming

Objects divide the solution space into logical chunks of state and behavior. Often, you need groups of your objects to perform their behavior simultaneously, as independent subtasks that make up the whole.

Each of these independent subtasks is called a *thread*, and you program as if each thread runs by itself and has the CPU to itself. Some underlying mechanism is actually dividing up the CPU time for you, but in general, you don't have to think about it, which makes programming with multiple threads a much easier task.

A *process* is a self-contained running program with its own address space. In C#, each application runs in its own process. A *multitasking* operating system is capable of running more than one process (program) at a time, while making it look like each one is chugging along on its own, by periodically providing CPU cycles to each thread within each process. A thread is a single sequential flow of control within a process (a "thread of execution"). A single process can thus have multiple concurrently executing threads.

There are many possible uses for multithreading, but in general, you'll have some part of your program tied to a particular calculation or resource, and you don't want to hang up the rest of your program because of that. So you create a thread associated with that calculation or resource and let it run independently of the main program. A good example is a "cancel" button to stop a lengthy calculation —you don't want to be forced to poll the cancel button in every piece of code you write in your program and yet you want the cancel button to be responsive, as if you *were* checking it regularly. In fact, one of the most immediately compelling reasons for multithreading is to produce a responsive user interface.

Responsive user interfaces

As a starting point, consider a program that performs some CPU-intensive operation and thus ends up ignoring user input and being unresponsive. This one simply covers the display in a random patchwork of red spots:

```
//:c16:Counter1.cs
// A non-responsive user interface.
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

class Counter1 : Form {
  int numberToCountTo;

  Counter1(int numberToCountTo) {
    this.numberToCountTo = numberToCountTo;
    ClientSize =
      new System.Drawing.Size(500, 300);
    Text = "Nonresponsive interface";

    Button start = new Button();
    start.Text = "Start";
    start.Location = new Point(10, 10);
    start.Click +=
      new EventHandler(StartCounting);

    Button onOff = new Button();
    onOff.Text = "Toggle";
    onOff.Location = new Point(10, 40);
    onOff.Click += new EventHandler(StopCounting);

    Controls.AddRange(
      new Control[]{ start, onOff});
  }

  public void StartCounting(
    Object sender, EventArgs args) {
    Rectangle bounds = Screen.GetBounds(this);
    int width = bounds.Width;
    int height = bounds.Height;

    Graphics g = this.CreateGraphics();
    using(g){
      Pen pen = new Pen(Color.Red, 1);
```

```
      Random rand = new Random();
      runFlag = true;
      for (int i = 0; runFlag
        && i < numberToCountTo; i++) {
        //Do something mildly time-consuming
        int x = rand.Next(width);
        int y = rand.Next(height);
        g.DrawRectangle(pen, x, y, 1, 1);
        Thread.Sleep(10);
      }
    }
  }
  bool runFlag = true;

  public void StopCounting(
    Object sender, EventArgs args){
    runFlag = false;
  }

  public static void Main() {
    Application.Run(new Counter1(10000));
  }
} ///:~
```

At this point, the graphics code should be reasonably familiar from Chapters 14 and 15, except that instead of painting the display in **OnPaint( )**, we use **Control.CreateGraphics( )** to stay busy: The method loops **numberToCountTo** times, picks a random spot on the form, and draws a 1-unit rectangle there.

Part of the loop inside **startCounting( )** calls **Thread.Sleep( )**. **Thread.Sleep( )** immediately pauses the currently executing thread for some amount of milliseconds. Regardless of whether you're explicitly using threads, you can produce the current thread used by your program with **Thread** and the static **Sleep( )** method.

When the **Start** button is pressed, **startCounting( )** is invoked. On examining **startCounting( )**, you might think that it should allow multithreading because it goes to sleep. That is, while the method is asleep, it seems like the CPU could be busy monitoring other button presses. But it turns out that the real problem is that **startCounting( )** doesn't return until *after* it's finished, and this means that **stopCounting( )** is never called until it's too late for its behavior to be

meaningful. Since you're stuck inside **startCounting( )** for the first button press, the program can't handle any other events. (To get out, you must either wait until **startCounting( )** ends, or kill the process; the easiest way to do this is to press Control-C or to click a couple times to trigger Windows' "Program Not Responding" dialogue.)

The basic problem here is that **startCounting( )** needs to continue performing its operations, and at the same time it needs to return so that **stopCounting( )** can be activated and the user interface can continue responding to the user. But in a conventional method like **startCounting( )** it cannot continue *and* at the same time return control to the rest of the program. This sounds like an impossible thing to accomplish, as if the CPU must be in two places at once, but this is precisely the illusion that threading provides.

The thread model (and its programming support in C#) is a programming convenience to simplify juggling several operations at the same time within a single program. With threads, the CPU will pop around and give each thread some of its time. Each thread has the consciousness of constantly having the CPU to itself, but the CPU's time is actually sliced between all the threads. The exception to this is if your program is running on multiple CPUs. But one of the great things about threading is that you are abstracted away from this layer, so your code does not need to know whether it is actually running on a single CPU or many. Thus, threads are a way to create transparently scalable programs.

Threading reduces computing efficiency somewhat, but the net improvement in program design, resource balancing, and user convenience is often quite valuable. Of course, if you have more than one CPU, then the operating system can dedicate each CPU to a set of threads or even a single thread and the whole program can run much faster. Multitasking and multithreading tend to be the most reasonable ways to utilize multiprocessor systems.

.NET's threading model

Like most modern operating systems, Windows differentiates between *processes*, which separate applications, and *threads*, which are the low-level unit of flow control inside a process; .NET introduces the concept of an *Application Domain* as an intermediate container between the two. An **AppDomain,** in addition to being responsible for loading and executing assemblies, is responsible for a *managed process*. In the same way that .NET has a managed heap that includes a Garbage Collector, managed processes are somewhat safer than unmanaged processes; specifically, a managed process can guarantee that **finally** clauses are

executed, that **IDisposable**.**Dispose( )** is called appropriately, and that garbage collection works properly with objects referenced by the managed process.

At any time, a given **Thread** is managed by some **AppDomain**. The controlling **AppDomain** may very well change over time, so **Thread.GetDomain( )** is a method, not a property. The **AppDomain** may, conversely, be managing many threads but there is no easy way to determine this set given just an **AppDomain** reference. You do not usually need to interact with the **AppDomain** of a **Thread**, but it is important for remoting (not discussed in this book).

You must manage your **Thread** through five of its seven states of its life: **Unstarted**, **Running**, **WaitSleepJoin**, **Suspended**, and **Stopped**. The runtime will handle two intermediate states, **SuspendRequested** and **AbortRequested**. At any time, **Thread.ThreadState** will be one of these values.

Not all **Thread**s need to enter all these states, as the *state-transition diagram* in Figure 16-1 illustrates. State-transition diagrams are UML diagrams that are especially handy to illustrate complex object lifecycles, such as is done here (they can also be used as the basis for a *Finite-State-Machine* architecture, popular with embedded programmers and developers of bad game AI).

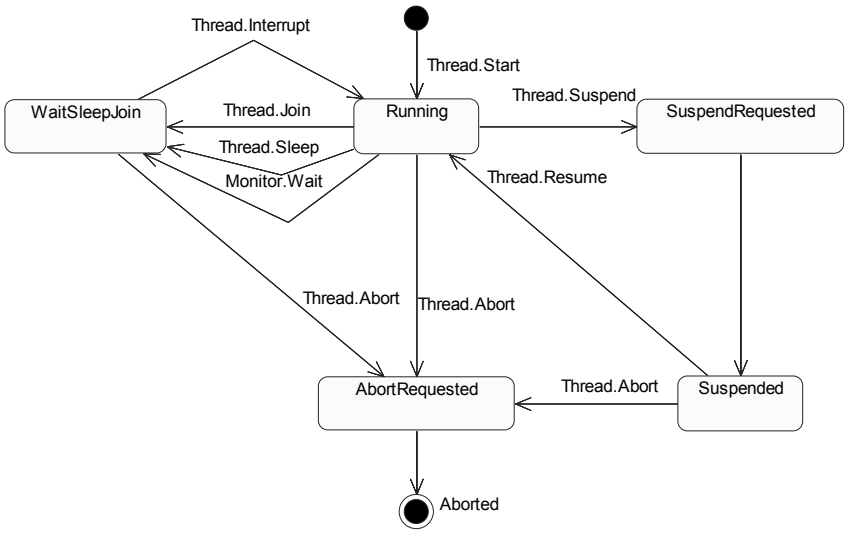


Figure 16-1: The lifecycle of a Thread

Running a thread

A .NET **Thread** must be constructed with a delegate of type **ThreadStart**. The **ThreadStart** delegate takes no arguments and returns nothing:

```
public delegate void ThreadStart();
```

Once constructed, **Thread.Start( )** will put the **Thread** in **ThreadState.Running** and the **ThreadStart** delegate will get called. If the **Thread** does not call the various methods shown in Figure #, and if no other **Thread** calls **Thread.Suspend( )** on your **Thread**, when the **ThreadStart( )** delegate returns, the **Thread** will call **Thread.Abort( )** on itself and eventually change to **ThreadState.Aborted**.

This sample shows the bare minimum needed to get some behavior out of a **Thread**.

```
//:c16:SecondaryThread.cs
//Demonstrates starting a thread
using System;
using System.Threading;

class SecondaryThread {
  static void MyThreadedExecutionBlock(){
    Console.WriteLine(
      "Executed from my thread:" +
      Thread.CurrentThread.ThreadState);
  }
  public static void Main(){
    ThreadStart myDelegate =
    new ThreadStart(MyThreadedExecutionBlock);
    Thread myThread = new Thread(myDelegate);
    Console.WriteLine(
      "Before Start(): " + myThread.ThreadState);
    myThread.Start();
    Console.WriteLine(
      "After Start():" + myThread.ThreadState);
  }
}///:~
```

The threading classes are in the **System.Threading** namespace, so we need to bring in those references with **using System.Threading**. The **Main( )** instantiates a **ThreadStart** delegate with the **MyThreadedExecutionBlock**

method. The delegate is passed to the **Thread( )** constructor and we write **myThread.ThreadState** to the console.

If you run this program, sometimes you may see this output:

```
Before Start(): Unstarted
Executed from my thread:Running
After Start():Stopped
```

which is exactly what you'd expect if, as with a normal method call, the execution of **Main( )** was suspended by the call to **Start( )** that in turn called **MyThreadedExecutionBlock( )**. However, most times you run this program, you'll see this:

```
Before Start(): Unstarted
After Start():Unstarted
Executed from my thread:Running
```

Precisely because **Main( )** is *not* suspended that long. **Main( )** *is* suspended during the call to **Start( )**, but **Thread.Start( )** simply triggers the start of the secondary thread, it does not wait for any behavior from it. The line that writes "After Start( )..." is written by the *application thread*; whether that line is written before **myThread** has begun processing or after its processing is a matter of a race between the main thread getting to the "After Start( )..." line before and the secondary thread getting to the "Executed from..." line.

Waiting for a thread to complete

It's appropriate that our very first multithreaded program demonstrates a *race condition* defect. When programming with threads, you must never assume the order in which threads will reach certain blocks of code. If we wanted to ensure that the secondary thread had completed, we might try this:

```
//:c16:BusyWait.cs
//A bad way to synchronize threads
using System;
using System.Threading;

class BusyWait {
  static bool secondaryThreadFinished = false;

  static void MyThreadedExecutionBlock(){
    Console.WriteLine(
      "Executed from my thread:" +
```

```
      Thread.CurrentThread.ThreadState);
    secondaryThreadFinished = true;
  }
  public static void Main(){
    ThreadStart myDelegate =
    new ThreadStart(MyThreadedExecutionBlock);
    Thread myThread = new Thread(myDelegate);
    Console.WriteLine(
      "Before Start(): " + myThread.ThreadState);
    myThread.Start();
    int iSpins = 0;
    //busy wait
    while (secondaryThreadFinished == false) {
      iSpins++;
    };
    Console.WriteLine(
      "After Start():" + myThread.ThreadState);
    Console.WriteLine("Spun {0} times", iSpins);
  }
}///:~
```

Here, **bool secondaryThreadFinished** is a *condition variable* that is used to communicate between threads. After **Main( )** calls **myThread.Start( )**, it spins in a tight loop, checking to see if the condition has been satisfied. This is what is called a *busy-wait*. The **iSpins** variable is not necessary for the logic of the program, but is used to illustrate why busy-waits are bad ideas. This program *will* have the desired behavior of delaying the writing of "After Start( )..." until after the secondary thread has completed, but you may be shocked at the value of **iSpins**. This value represents a lot of wasted CPU effort.

Instead of a busy-wait, changing the loop to:

```
   while(secondaryThreadFinished == false){
        //Sleep for 1 ms.
        Thread.Sleep(1);
        iSpins++;
      };
```

will have a dramatic effect on the final value of **iSpins** (it will probably reduce it to 1). The static method **Thread.Sleep( )** takes an integer representing a time in milliseconds or a **TimeSpan** and places the current thread (that is, the **Thread** that is executing the line of code) into the **WaitSleepJoin** state for that

amount of time. This suspension is done at the OS level and allows the CPU to be used efficiently.

If you pass a value of 0 to **Thread.Sleep( ),** you suspend the current thread so that other waiting threads may execute, but you are not requesting an actual pause in the current thread's processing. If you pass a value of **Timeout.Infinite,** you are requesting that the **Thread** be put to sleep forever; we'll cover how to get out of that situation with **Thread.Interrupt( )** a little later.

Multiple threads in action

The following example creates any number of threads that it keeps track of by assigning each thread a unique number, generated with a **static** variable. The delegated behavior is in the **Run( )** method, which is overridden to count down each time it passes through its loop and finishes when the count is zero (at the point when the delegated method returns, the thread is terminated).

```
//:c16: SimpleThreading.cs
// Demonstrates scheduling
using System;
using System.Threading;

public class SimpleThreading {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;

  public SimpleThreading() {
    Console.WriteLine(
      "Making " + threadNumber);
  }

  public void Run() {
    while (true) {
      Console.WriteLine("Thread " +
        threadNumber + "(" + countDown + ")");
      if (--countDown == 0) return;
    }
  }

  public static void Main() {
```

```
    for (int i = 0; i < 5; i++) {
      SimpleThreading st = new SimpleThreading();
      Thread aThread =
      new Thread( new ThreadStart(st.Run));
      aThread.Start();
    }
    Console.WriteLine("All Threads Started");
  }
} ///:~
```

The **ThreadStart** delegate method (often called **Run( )**) virtually always has some kind of loop that continues until the thread is no longer necessary, so you must establish the condition on which to break out of this loop (or, in the case above, simply **return** from **Run( )**). Often, **Run( )** is cast in the form of an infinite loop, which means that, barring some external factor that causes **Run( )** to terminate, it will continue forever.

In **Main( )** you can see a number of threads being created and run. The **Start( )** method in the **Thread** class performs the initialization for the thread and then calls **Run( )**. So the steps are: The constructor is called to build the object that will do the work, the **ThreadStart** delegate is given the name of the working function, the **ThreadStart** is passed to a newly created **Thread**, then **Start( )** configures the thread and calls the delegated function — **Run( )**. If you don't call **Start( )**, the thread will never be started.

The output for one run of this program (it will be different from one run to another) is:

```
Making 1
Making 2
Making 3
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 1(1)
Making 4
Making 5
All Threads Started
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
```

```
Thread 2(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 3(5)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

You'll notice that nowhere in this example is **Thread.Sleep( )** called, and yet the output indicates that each thread gets a portion of the CPU's time in which to execute. This shows that **Sleep( )**, while it relies on the existence of a thread in order to execute, is not involved with either enabling or disabling threading. It's simply another method.

You can also see that the threads are not run in the order that they're created. In fact, the order that the CPU attends to an existing set of threads is indeterminate, unless you go in and adjust the **Priority** property of the thread.

When **Main( )** creates the **Thread** objects it isn't capturing the references for any of them. This is where the managed process capability of the **AppDomain** comes into play. An ordinary object with no references to it would be fair game for garbage collection, but not a **Thread** in a managed process. As long as the **Thread** has not entered **ThreadState.Stopped**, it will not be garbage collected.

Threading for a responsive interface

Now it's possible to solve the problem in **Counter1.cs** with a thread. The trick is to make the working method —that is, the loop that's inside **StopCounting( )**— a delegate of a **Thread**. When the user presses the **start** button, the thread is started, but then the *creation* of the thread completes, so even though the thread is running, the main job of the program (watching for and responding to user-interface events) can continue. Here's the solution:

```
//:c16:Counter2.cs
// A responsive user interface.
using System;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

class Counter2 : Form {
  int numberToCountTo;

  Counter2(int numberToCountTo) {
    this.numberToCountTo = numberToCountTo;
    ClientSize = new Size(500, 300);
    Text = "Responsive interface";

    Button start = new Button();
    start.Text = "Start";
    start.Location = new Point(10, 10);
    start.Click +=
      new EventHandler(StartCounting);

    Button onOff = new Button();
    onOff.Text = "Toggle";
    onOff.Location = new Point(10, 40);
    onOff.Click +=
      new EventHandler(StopCounting);

    this.Closing +=
      new CancelEventHandler(StopCountingIfClosing);

    Controls.AddRange(
      new Control[]{ start, onOff});
  }

  public void StartCounting(
    Object sender, EventArgs args) {
    ThreadStart del =
      new ThreadStart(paintScreen);
    Thread t = new Thread(del);
    t.Start();
```

```
  }

  public void paintScreen(){
    Rectangle bounds = Screen.GetBounds(this);
    int width = bounds.Width;
    int height = bounds.Height;

    Graphics g = CreateGraphics();
    using(g){
      Pen pen = new Pen(Color.Red, 1);

      Random rand = new Random();
      runFlag = true;
      for (int i = 0; runFlag
        && i < numberToCountTo; i++) {
        //Do something mildly time-consuming
        int x = rand.Next(width);
        int y = rand.Next(height);
        g.DrawRectangle(pen, x, y, 1, 1);
        Thread.Sleep(10);
      }
    }
  }

  bool runFlag = true;

  public void StopCounting(
    object sender, EventArgs args){
    runFlag = false;
  }

  public void StopCountingIfClosing(object sender,
    CancelEventArgs ea){
    runFlag = false;
  }

  public static void Main() {
    Application.Run(new Counter2(10000));
  }
} ///:~
```

Counter2 is a straightforward program, whose only job is to set up and maintain the user interface. But now, when the user presses the **start** button, the event-handling code does not do the time-consuming work. Instead a thread is created and started, and then the **Counter2** interface can continue to respond to events.

When you press the **onOff** button it toggles the **runFlag** condition variable inside the **Counter2** object. Then, when the "worker" thread calls **PaintScreen( )**, it can look at that flag and decide whether to continue or stop. Pressing the **onOff** button produces an apparently instant response. Of course, the response isn't really instant, not like that of a system that's driven by interrupts. The painting stops only when the thread has the CPU and notices that the flag has changed.

If the thread painting the screen is running when the form is closed, the program will throw a fairly cryptic **ExternalException** with the explanation that "a generic error occurred in GDI+." To get rid of this, we add another delegate **StopCountingIfClosing** and attach it to the **Counter2** form's **Closing** event. This also requires us to add **using System.ComponentModel** at the top of the program.

Interrupting a sleeping Thread

In **Counter2**, the loop within **PaintScreen( )** calls **Thread.Sleep( )** to sleep for 20 milliseconds. What if some crisis happened and even that was too long? Or, to return to something we discussed earlier, what if **Thread.Sleep( )** was called with the value of **Timeout.Infinite**, would we be doomed to face another non-responsive interface? The answer is that in both situations, we can use **Thread.Interrupt( )** to force the **Thread** to wake.

If you refer to Figure 16-1, you'll see that to get out of **ThreadState.WaitSleepJoin** and back to **ThreadState.Running** requires someone to call **Thread.Interrupt( )**. When you call **Thread.Sleep( )**, the call is done by the runtime when the time's up. But you can do it yourself if you want to interrupt a sleeping thread (the method name **Thread.Interrupt( )** may be a little counter-intuitive at first, as stopping or pausing an active thread is also sometimes called "interrupting the thread." If you just repeat "You can **Interrupt** a **Sleep**ing **Thread**" enough times, you'll get used to it.)

When you call **Thread.Interrupt( )**, a **ThreadInterruptedException** is thrown from the method you called that put the **Thread** in the **WaitSleepJoin** state (in this case, **Thread.Sleep( )**). If you do not catch this exception, it will propagate out of your **ThreadStart** delegate and your **Thread** will abort.

This example shows a **Cat** which has a **CatNap( )** method that is used as a **ThreadStart** delegate. The **Cat** is interested in a **Mouse** that is not hiding in the walls. The **Cat** wakes up every 10 seconds for just long enough to determine if the mouse is in the room, if not, the **Cat** goes to sleep again. The **Mouse** class has a method **SneakAbout( )** which is also to be used as an instance of **ThreadStart**. The **Mouse** has a 20% chance of moving from hole to room and vice versa and, if in the room, it has a 10% chance of squeaking. If it squeaks, it will call **Thread.Interrupt( )** on the **theCat Thread**.

```
//:c16:CatNap.cs
//Demonstrates Thread.Interrupt
using System;
using System.Threading;

class Cat {
  public void CatNap(){
    try {
      while (prey.InHole == true) {
        Console.WriteLine("Going back to sleep");
        //Sleep for 10 seconds
        Thread.Sleep(10000);
        Console.WriteLine(
          "Waking up @" + DateTime.Now);
      }
    } catch (ThreadInterruptedException) {
      Console.WriteLine(
        "Awakened @" + DateTime.Now);
    }
    Console.WriteLine("Chasing mouse");
  }

  Mouse prey;
  public Mouse Prey{
    set { prey = value;}
    get { return prey;}
  }
}

class Mouse {
  static Random r = new Random();
  Thread theCat;
```

```
  bool inHole = true;
  public bool InHole{
    set { inHole = value;}
    get { return inHole;}
  }

  internal Mouse(Thread theCat){
    this.theCat = theCat;
  }

  public void SneakAbout(){
    while(theCat.ThreadState != ThreadState.Stopped) {
      if (inHole) {
        Console.WriteLine("In hole");
        Thread.Sleep(1000);
        if (r.NextDouble() < .2) {
          Console.WriteLine("Going for a walk");
          inHole = false;
        }
      } else {
        Console.WriteLine("In room");
        if (r.NextDouble() < .2) {
          Console.WriteLine("Getting to shelter");
          inHole = true;
        } else {
          if (r.NextDouble() < .1) {
            Console.WriteLine("Squeak! Squeak!");
            theCat.Interrupt();
          }
          Thread.Sleep(1000);
        }
      }
    }
  }
}

class TomAndJerry {
  public static void Main(){
    Cat tom = new Cat();
    ThreadStart tomDel =
      new ThreadStart(tom.CatNap);
```

```
    Thread tomThread = new Thread(tomDel);
    Mouse jerry = new Mouse(tomThread);
    tom.Prey = jerry;
    ThreadStart jerryDel =
      new ThreadStart(jerry.SneakAbout);
    Thread jerryThread = new Thread(jerryDel);
    jerryThread.Start();
    tomThread.Start();
```

This is a very good example, but I think it would be even more clear if the output was prepended with the name of the thread it's coming from (i.e. "Jerry: In Room" or "Tom: Chasing mouse")

```
  }
}///:~
```

When you run this, sometimes it will end when the mouse is quietly in the room and the cat happens to wake up. Other times, it will end because **Thread.Interrupt( )** immediately wakes the cat, triggering the exception handler. If the **ThreadInterruptedException** was not handled in the cat's **ThreadStart** method **CatNap( )**, the call to **myCat.Interrupt( )** would just terminate the **myCat** thread silently.

Thread.Join( ) waits for another thread to end

If you thought **Thread.Interrupt( )** was a counterintuitive name, **Thread.Join( )** is downright inexplicable. **Thread.Join( )** blocks the calling thread until the **Thread** on which **Join( )** has been called terminates. Why this is called "joining" the other thread is a mystery.[1]

Strange name aside, **Thread.Join( )** is very useful, as waiting for another thread to end is probably one of the most commonly desired behaviors. Going back to the **SecondaryThread** example, **Thread.Join( )** allows us to eliminate both the condition variable and the loop.

```
//:c16:JoinDemo.cs
//Waiting for another thread to end
using System;
```

---

1 This is not a C# or .NET issue; this use of the word join is traditional in the world of multithreading. Perhaps you can think of "boards are Joined end-to-end" as a mnemonic.

```
using System.Threading;

class JoinDemo {
  static void MyThreadedExecutionBlock(){
    Console.WriteLine(
      "Executed from my thread:" +
      Thread.CurrentThread.ThreadState);
  }
  public static void Main(){
    ThreadStart myDelegate =
    new ThreadStart(MyThreadedExecutionBlock);
    Thread myThread = new Thread(myDelegate);
    Console.WriteLine(
      "Before Start(): " + myThread.ThreadState);
    myThread.Start();
    //Put main application thread in WaitSleepJoin
    //until myThread becomes ThreadState.Stopped
    myThread.Join();
    Console.WriteLine(
      "After Start():" + myThread.ThreadState);
  }
}///:~
```

The line **myThread.Join( )** is executed by the main application thread. When executed, the main application thread changes to **ThreadState.WaitSleepJoin** until **myThread** changes to **ThreadState.Stopped**. (However, if someone had a reference to the main thread of execution and called **Thread.Interrupt( )** on it, a **ThreadInterruptedException** would be thrown from inside the **myThread.Join( )** call. See the preceding section for an explanation.)

If you do not trust that the other thread will end, you can pass an **int** millisecond value or **TimeSpan** to **Join( )**. If the other thread hasn't ended before the timeout elapses, the current thread will restart.

```
//:c16:JoinTimeout.cs
//Demonstrates a Join that times out
using System;
using System.Threading;

class JoinTimeOut {
  static public void DoesNotEnd(){
```

```
    Thread.Sleep(Timeout.Infinite);
  }

  public static void Main(){
    ThreadStart del = new ThreadStart(DoesNotEnd);
    Thread nonEndingThread = new Thread(del);
    nonEndingThread.IsBackground = true;
    nonEndingThread.Start();
    //Timeout after 3 seconds
    nonEndingThread.Join(3000);
    Console.WriteLine("Other thread is: " +
      nonEndingThread.ThreadState);
  }
}///:~
```

If you comment out the line

```
nonEndingThread.IsBackground = true;
```

the program will continue to run even after **Main( )** has exited. Every **Thread** is either a background thread or a foreground thread; an application only exits when all foreground threads have stopped. For Java programmers, this is the equivalent of the **Thread.isDaemon( )**.

Sharing limited resources

The state-transition diagram in Figure 16-1 shows one more method that can put a **Thread** in **ThreadState.WaitSleepJoin**: **Monitor.Wait( )**. Before we can discuss this method, though, we'll have to introduce some more multithreading complexity.

You can think of a single-threaded program as one lonely entity moving around through your problem space and doing one thing at a time. Because there's only one entity, you never have to think about the problem of two entities trying to use the same resource at the same time, like two people trying to park in the same space, walk through a door at the same time, or even talk at the same time.

With multithreading, things aren't lonely anymore, but you now have the possibility of two or more threads trying to use the same limited resource at once. Colliding over a resource must be prevented or else you'll have two threads trying to access the same bank account at the same time, print to the same printer, or adjust the same valve, etc.

Improperly accessing resources

Consider a variation on the counters that have been used so far in this chapter. In the following example, each thread contains two counters that are incremented and displayed inside **Run( )**. In addition, there's another thread of class **Watcher** that is watching the counters to see if they're always equivalent. This seems like a needless activity, since looking at the code it appears obvious that the counters will always be the same. But that's where the surprise comes in. Here's the first version of the program:

```
//:c16:Sharing1.cs
// Problems with resource sharing while threading.
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

public class Sharing1 : Form {
  private TextBox accessCountBox = new TextBox();
  private Button start = new Button();
  private Button watch = new Button();

  private int accessCount = 0;
  public void IncrementAccess() {
    accessCount++;
    accessCountBox.Text = accessCount.ToString();
  }

  private int numCounters = 12;
  private int numWatchers = 15;

  private TwoCounter[] s;

  public Sharing1() {
    ClientSize = new Size(450, 480);
    Panel p = new Panel();
    p.Size = new Size(400, 50);

    start.Click +=
      new EventHandler(StartAllThreads);
    watch.Click +=
      new EventHandler(StartAllWatchers);
```

```
    accessCountBox.Text = "0";
    accessCountBox.Location = new Point(10, 10);
    start.Text = "Start threads";
    start.Location = new Point(110, 10);
    watch.Text = "Begin watching";
    watch.Location = new Point(210, 10);

    p.Controls.Add(start);
    p.Controls.Add(watch);
    p.Controls.Add(accessCountBox);

    s = new TwoCounter[numCounters];
    for (int i = 0; i < s.Length; i++) {
      s[i] = new TwoCounter(
        new TwoCounter.IncrementAccess(
          IncrementAccess));
      s[i].Location =
        new Point(10, 50 + s[i].Height * i);
      Controls.Add(s[i]);
    }

    this.Closed += new EventHandler(StopAllThreads);

    Controls.Add(p);
  }

  public void StartAllThreads(
    Object sender, EventArgs args) {
    for (int i = 0; i < s.Length; i++)
      s[i].Start();
  }

  public void StopAllThreads(
    Object sender, EventArgs args){
    for (int i = 0; i < s.Length; i++) {
      if (s[i] != null) {
        s[i].Stop();
      }
    }
  }
```

```
  public void StartAllWatchers(
    Object sender, EventArgs args) {
    for (int i = 0; i < numWatchers; i++)
      new Watcher(s);
  }

  public static void Main(string[] args) {
    Sharing1 app = new Sharing1();
    if (args.Length > 0) {
      app.numCounters = SByte.Parse(args[0]);
      if (args.Length == 2) {
        app.numWatchers = SByte.Parse(args[1]);
      }
    }
    Application.Run(app);
  }
}

class TwoCounter : Panel {
  private bool started = false;
  private Label t1;
  private Label t2;
  private Label lbl;
  private Thread t;

  private int count1 = 0, count2 = 0;
  public delegate void IncrementAccess();
  IncrementAccess del;

  // Add the display components
  public TwoCounter(IncrementAccess del) {
    this.del = del;

    this.Size = new Size(350, 30);
    this.BorderStyle = BorderStyle.Fixed3D;
    t1 = new Label();
    t1.Location = new Point(10, 10);
    t2 = new Label();
    t2.Location = new Point(110, 10);
    lbl = new Label();
```

```
    lbl.Text = "Count1 == Count2";
    lbl.Location = new Point(210, 10);
    Controls.AddRange(new Control[]{t1, t2, lbl});

    //Initialize the Thread
    t = new Thread(new ThreadStart(Run));
    t.IsBackground = true;
  }
  public void Start() {
    if (!started) {
      started = true;
      t.Start();
    }
  }

  public void Stop(){
    t.Abort();
  }
  public void Run() {
    while (true) {
      t1.Text = (++count1).ToString();
      t2.Text = (++count2).ToString();
      Thread.Sleep(500);
    }
  }
  public void SynchTest() {
    del();
    if (count1 != count2)
      lbl.Text = "Unsynched";
  }
}

class Watcher {
  TwoCounter[] s;

  public Watcher(TwoCounter[] s) {
    this.s = s;
    Thread t = new Thread(new ThreadStart(Run));
    t.IsBackground = true;
    t.Start();
  }
```

```
  public void Run() {
    while (true) {
      for (int i = 0; i < s.Length; i++)
        s[i].SynchTest();
      Thread.Sleep(500);
    }
  }
}///:~
```

As before, each counter contains its own display components: two text fields and a label that initially indicates that the counts are equivalent. These components are added to the panel of the **Sharing1** object in the **Sharing1** constructor.

Because a **TwoCounter** thread is started via a button press by the user, it's possible that **Start( )** could be called more than once. It's illegal for **Thread.Start( )** to be called more than once for a thread (an exception is thrown). You can see the machinery to prevent this in the **started** flag and the **Start( )** method.

The **accessCountBox** in **Sharing1** keeps track of how many total accesses have been made on all **TwoCounter** threads. One way to do this would have been to have a static property that each **TwoCounter** could have incremented during **SynchTest( )**. Instead, we declared an **IncrementAccess( )** delegate within **TwoCounter** that **Sharing1** provides as a parameter to the **TwoCounter** constructor.

In **Run( )**, **count1** and **count2** are incremented and displayed in a manner that would seem to keep them identical. Then **Sleep( )** is called; without this call the UI becomes unresponsive because all the CPU time is being consumed within the loops.

The **SynchTest( )** calls its **IncrementAccess** delegate and then performs the apparently superfluous activity of checking to see if **count1** is equivalent to **count2**; if they are not equivalent it sets the label to "Unsynched" to indicate this.

The **Watcher** class is a thread whose job is to call **SynchTest( )** for all of the **TwoCounter** objects that are active. It does this by stepping through the array of **TwoCounters** passed to it by the **Sharing1** object. You can think of the **Watcher** as constantly peeking over the shoulders of the **TwoCounter** objects.

Sharing1 contains an array of **TwoCounter** objects that it initializes in its constructor and starts as threads when you press the "Start Threads" button.

Later, when you press the "Begin Watching" button, one or more watchers are created and free to spy upon the unsuspecting **TwoCounter** threads.

By changing the **numCounters** and **numWatchers** values, which you can do at the command-line, you'll change the behavior of the program.

Here's the surprising part. In **TwoCounter.Run( )**, the infinite loop is just repeatedly passing over the adjacent lines:

```
t1.Text = (++count1).ToString();
t2.Text = (++count2).ToString();
```

(as well as sleeping, but that's not important here). When you run the program, however, you'll discover that **count1** and **count2** will be observed (by the **Watcher**s) to be unequal at times! This is because of the nature of threads—they can be suspended at any time. So at times, the suspension occurs *between* the time **count1** and **count2** are incremented, and the **Watcher** thread happens to come along and perform the comparison at just this moment, thus finding the two counters to be different.

This example shows a fundamental problem with using threads. You never know when a thread might be run. Imagine sitting at a table with a fork, about to spear the last piece of food on your plate and as your fork reaches for it, the food suddenly vanishes (because your thread was suspended and another thread came in and stole the food). That's the problem that you're dealing with. Any time you rely on the state of an object being consistent, and that state can be manipulated by a different thread, you are vulnerable to this type of problem. This is another manifestation of a *race condition* defect (because your program's proper functioning is dependent on its thread winning the "race" to the resource). This type of bug (and all bugs relating to threading) is difficult to track down, as they will often slip under the radar of your unit testing code and appear and disappear depending on load, hardware and operating system differences, and the whimsy of the fates.

Preventing this kind of collision is simply a matter of putting a lock on a resource when one thread is relying on that resource. The first thread that accesses a resource locks it, and then the other threads cannot access that resource until it is unlocked, at which time another thread locks and uses it, etc. If the front seat of the car is the limited resource, the child who shouts "Dibs!" asserts the lock.

Using Monitor to prevent collisions

Real-world programs have to share many types of resources – network sockets, database connections, sound channels, etc. By far the most common collisions,

though, occur when some threads are changing the states of objects and other threads are relying on the state being consistent. This is the case with our **TwoCounter** and **Watcher** objects, where a thread controlled by **TwoCounter** increments the **count1** and **count2** variables, while a thread controlled by **Watcher** checks these variables for consistency.

The **Monitor** class helps prevent collisions over object state. Every reference type in C# has an associated "synchronization block" object which maintains a lock for that object and a queue of threads waiting to access the lock. The **Monitor** class is the public interface to the behind-the-scenes sync block implementation.

The method **Monitor.Enter(object o)** acts as gatekeeper – when a thread executes this line, the synchronization block for **o** is checked; if no one currently has the lock, the thread gets the lock and processing continues, but if another thread has already acquired the lock, the thread waits, or "blocks," until the lock becomes available. When the critical section of code has been executed, the thread should call **Monitor.Exit(object o)** to release the lock. The next blocking thread will then be given a chance to obtain the lock.

Making value types, which are created on the stack or inline, **Monitor**-able would be *hugely* inefficient, so you can only use **Monitor** on reference types. If you incorrectly use a value type as an argument to **Monitor.Enter( ),** it will throw a **SynchronizationLockException** at runtime. The current version of the runtime diagnoses an attempt to lock a value type with this somewhat misleading message: "Object synchronization method was called from an unsynchronized block of code."

Because you virtually always want to release the lock on a thread at the end of a critical section, even when throwing an **Exception**, calls to **Monitor.Enter( )** and **Monitor.Exit( )** are usually wrapped in a try block:

```
try{
    Monitor.Enter(o);
    //critical section
}finally{
    Monitor.Exit(o);
}
```

Synchronizing the counters

Armed with this technique it appears that the solution is at hand: We'll use the **Monitor** class to synchronize access to the counters. The following example is

the same as the previous one, with the addition of **Monitor.Enter/Exit** calls at the two critical sections:

```
//:c16:Sharing2.cs
// Problems with resource sharing while threading.
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

public class Sharing2 : Form {
  private TextBox accessCountBox = new TextBox();
  private Button start = new Button();
  private Button watch = new Button();

  private int accessCount = 0;
  public void IncrementAccess() {
    accessCount++;
    accessCountBox.Text = accessCount.ToString();
  }

  private int numCounters = 12;
  private int numWatchers = 15;

  private TwoCounter[] s;

  public Sharing2() {
    ClientSize = new Size(450, 480);
    Panel p = new Panel();
    p.Size = new Size(400, 50);

    start.Click +=
      new EventHandler(StartAllThreads);
    watch.Click +=
      new EventHandler(StartAllWatchers);

    accessCountBox.Text = "0";
    accessCountBox.Location = new Point(10, 10);
    start.Text = "Start threads";
    start.Location = new Point(110, 10);
    watch.Text = "Begin watching";
```

```
    watch.Location = new Point(210, 10);

    p.Controls.Add(start);
    p.Controls.Add(watch);
    p.Controls.Add(accessCountBox);

    s = new TwoCounter[numCounters];
    for (int i = 0; i < s.Length; i++) {
      s[i] = new TwoCounter(
        new TwoCounter.IncrementAccess(
          IncrementAccess));
      s[i].Location =
        new Point(10, 50 + s[i].Height * i);
      Controls.Add(s[i]);
    }

    this.Closed += new EventHandler(StopAllThreads);

    Controls.Add(p);
  }

  public void StartAllThreads(
    Object sender, EventArgs args) {
    for (int i = 0; i < s.Length; i++)
      s[i].Start();
  }

  public void StopAllThreads(
    Object sender, EventArgs args){
    for (int i = 0; i < s.Length; i++) {
      if (s[i] != null) {
        s[i].Stop();
      }
    }
  }

  public void StartAllWatchers(
    Object sender, EventArgs args) {
    for (int i = 0; i < numWatchers; i++)
      new Watcher(s);
  }
```

```
  public static void Main(string[] args) {
    Sharing2 app = new Sharing2();
    if (args.Length > 0) {
      app.numCounters = SByte.Parse(args[0]);
      if (args.Length == 2) {
        app.numWatchers = SByte.Parse(args[1]);
      }
    }
    Application.Run(app);
  }
}

class TwoCounter : Panel {
  private bool started = false;
  private Label t1;
  private Label t2;
  private Label lbl;
  private Thread t;

  private int count1 = 0, count2 = 0;
  public delegate void IncrementAccess();
  IncrementAccess del;

  // Add the display components
  public TwoCounter(IncrementAccess del) {
    this.del = del;

    this.Size = new Size(350, 30);
    this.BorderStyle = BorderStyle.Fixed3D;
    t1 = new Label();
    t1.Location = new Point(10, 10);
    t2 = new Label();
    t2.Location = new Point(110, 10);
    lbl = new Label();
    lbl.Text = "Count1 == Count2";
    lbl.Location = new Point(210, 10);
    Controls.AddRange(new Control[]{t1, t2, lbl});

    //Initialize the Thread
    t = new Thread(new ThreadStart(Run));
```

```
    t.IsBackground = true;
  }
  public void Start() {
    if (!started) {
      started = true;
      t.Start();
    }
  }

  public void Stop(){
    t.Abort();
  }
  public void Run() {
    while (true) {
      try {
        Monitor.Enter(this);
        t1.Text = (++count1).ToString();
        t2.Text = (++count2).ToString();
        Thread.Sleep(0);
      } finally {
        Monitor.Exit(this);
      }
    }
  }
  public void SynchTest() {
    del();
    try {
      Monitor.Enter(this);
      if (count1 != count2) {
        lbl.Text = "Unsynched";
      }
    } finally {
      Monitor.Exit(this);
    }
  }
}

class Watcher {
  TwoCounter[] s;

  public Watcher(TwoCounter[] s) {
```

```
    this.s = s;
    Thread t = new Thread(new ThreadStart(Run));
    t.IsBackground = true;
    t.Start();
  }
  public void Run() {
    while (true) {
      for (int i = 0; i < s.Length; i++)
        s[i].SynchTest();
      Thread.Sleep(500);
    }
  }
}///:~
```

You'll notice that *both* **TwoCounter.Run( )** and **SynchTest( )** call **Monitor.Enter( )** and **Exit( ).** If you use the **Monitor** only on **Run( )**, the **Watcher** threads calling **SynchTest( )** will happily read the state of the **TwoCounter** even while the **TwoCounter** thread has entered the critical section and has placed the object in an inconsistent state by incrementing **Counter1** but not yet changing **Counter2**.

There's nothing magic about thread synchronization – it's the manipulation of the **TwoCounter** instance's sync block via the **Monitor** that does the work. All synchronization depends on programmer diligence: Every piece of code that can place an object in an inconsistent state or that relies on an object being in a consistent state must be wrapped in an appropriate block.

lock blocks – a shortcut for using Monitor

In order to save you some time typing, C# provides the **lock** keyword, which creates a guarded code block exactly equivalent to the

```
try{
   Monitor.Enter();
   …
}finally{
   Monitor.Exit();
}
```

idiom. Instead of all that typing, you use the lock(object) keyword and specify a code block to be protected:

```
lock(this){
```

```
    //critical section
}
```

Although **lock( )**'s form makes it appear to be a call to a base class method (implemented by **object** presumably), it's really just syntactic sugar. The choice of **lock** as a keyword may be a little misleading, in that you may expect that a "locked" object would be automatically thread-safe. This is not so; **lock** says that the *current* thread will act appropriately. If all threads follow the rules, things will work out for the best, but if another piece of code has a reference to the "locked" object, it may mistakenly manipulate the object without ever using **Monitor**. It's like taking a number for service at a bakery – a fine idea that breaks down as soon as someone misbehaves through laziness or ignorance. Java has a similar mechanism, but uses the keyword **synchronized**, which gives a better indication that success requires the cooperation of multiple objects and threads.

Another advantage of **lock** over **Monitor.Enter( )** is that the compiler will refuse to compile code that attempts to lock a value type.

Choosing what to monitor

Monitor.Enter( ) and **Exit( )** can be passed any object for synchronization. It is best to use **this** – an inability to use **this** for synchronization (perhaps because you have mutually exclusive types of inconsistency to guard against) is a "code smell" that indicates that your object may be trying to do too many things at once and may be best broken up into smaller classes. For instance, in the **TwoCounter** class, we could place **counter1** and **counter2** in an inner class, add thread-safe accessors and methods, and achieve our goal without ever locking the entire **TwoPanel** instance:

```
//:c16:Sharing3.cs
// Refactoring for finer granularity
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

public class Sharing3 : Form {
  private TextBox aCount = new TextBox();
  private Button start = new Button();
  private Button watch = new Button();

  private int accessCount = 0;
```

```
public void IncrementAccess() {
  accessCount++;
  aCount.Text = accessCount.ToString();
}

private int numCounters = 12;
private int numWatchers = 15;

private TwoCounter[] s;
private Watcher[] w;

public Sharing3() {
  ClientSize = new Size(450, 480);
  Panel p = new Panel();
  p.Size = new Size(400, 50);

  start.Click +=
    new EventHandler(StartAllThreads);
  watch.Click +=
    new EventHandler(StartAllWatchers);

  aCount.Text = "0";
  aCount.Location = new Point(10, 10);
  start.Text = "Start threads";
  start.Location = new Point(110, 10);
  watch.Text = "Begin watching";
  watch.Location = new Point(210, 10);

  p.Controls.Add(start);
  p.Controls.Add(watch);
  p.Controls.Add(aCount);

  s = new TwoCounter[numCounters];
  for (int i = 0; i < s.Length; i++) {
    s[i] = new TwoCounter(
      new TwoCounter.IncrementAccess(
        IncrementAccess));
    s[i].Location =
      new Point(10, 50 + s[i].Height * i);
    Controls.Add(s[i]);
  }
```

```
    this.Closed +=
      new EventHandler(StopAllThreads);

    Controls.Add(p);
  }

  public void StartAllThreads(
    Object sender, EventArgs args) {
    for (int i = 0; i < s.Length; i++)
      s[i].Start();
  }

  public void StopAllThreads(
    Object sender, EventArgs args){
    if (s != null) {
      for (int i = 0; i < s.Length; i++) {
        if (s[i] != null) {
          s[i].Stop();
        }
      }
    }
    if (w != null) {
      for (int i = 0; i < w.Length; i++) {
        if (w[i] != null) {
          w[i].Stop();
        }
      }
    }
  }

  public void StartAllWatchers(
    Object sender, EventArgs args) {
    w = new Watcher[numWatchers];
    for (int i = 0; i < numWatchers; i++)
      w[i] = new Watcher(s);
  }

  public static void Main(string[] args) {
    Sharing3 app = new Sharing3();
    if (args.Length > 0) {
```

```
      app.numCounters = SByte.Parse(args[0]);
      if (args.Length == 2) {
        app.numWatchers = SByte.Parse(args[1]);
      }
    }
    Application.Run(app);
  }
}

class TwoCounter : Panel {
  private bool started = false;
  private Label t1;
  private Label t2;
  private Label lbl;
  private Thread t;

  class Counter {
    private int c1 = 0;
    private int c2 = 0;
    public void Increment(){
      lock(this){
        ++c1;
        ++c2;
      }
    }
    public int Count1{
      get{
        lock(this){ return c1;}
      }
    }
    public int Count2{
      get{
        lock(this){ return c2;}
      }
    }
  }
  private Counter counter = new Counter();
  public delegate void IncrementAccess();
  IncrementAccess del;

  // Add the display components
```

```
  public TwoCounter(IncrementAccess del) {
    this.del = del;

    this.Size = new Size(350, 30);
    this.BorderStyle = BorderStyle.Fixed3D;
    t1 = new Label();
    t1.Location = new Point(10, 10);
    t2 = new Label();
    t2.Location = new Point(110, 10);
    lbl = new Label();
    lbl.Text = "Count1 == Count2";
    lbl.Location = new Point(210, 10);
    Controls.AddRange(
      new Control[]{t1, t2, lbl});

    //Initialize the Thread
    t = new Thread(new ThreadStart(Run));
    t.IsBackground = true;
  }
  public void Start() {
    if (!started) {
      started = true;
      t.Start();
    }
  }

  public void Stop(){
    t.Abort();
  }
  public void Run() {
    while (true) {
      counter.Increment();
      t1.Text =
        counter.Count1.ToString();
      t2.Text =
        counter.Count2.ToString();
      Thread.Sleep(500);
    }
  }
  public void SynchTest() {
    del();
```

```
    if (counter.Count1 != counter.Count2) {
      lbl.Text = "Unsynched";
    }
  }
}

class Watcher {
  TwoCounter[] s;
  Thread t;

  public Watcher(TwoCounter[] s) {
    this.s = s;
    t = new Thread(new ThreadStart(Run));
    t.IsBackground = true;
    t.Start();
  }
  public void Run() {
    while (true) {
      for (int i = 0; i < s.Length; i++)
        s[i].SynchTest();
      Thread.Sleep(500);
    }
  }
  public void Stop(){
    t.Abort();
  }
}///:~
```

The **Counter** class, an inner class of **TwoPanel** is now "thread-safe," by removing any chance that an external object can place it in an inconsistent state. Incrementing the counter integers is done inside the **Increment** method, with a **lock**ed critical section, and access to the integers is done via the **Count1** and **Count2** properties, which also synchronize against the Monitor to ensure that they cannot be read until the **Increment** critical section has exited (and, undesirably, also ensure that **Count1** and **Count2** cannot be read simultaneously by two different threads – a small penalty typical of the design decisions made when developing multithreaded apps).

Sometimes, locking **this** doesn't seem like the right idea. When an object contains some resource, and especially when that resource is a container of some sort, such as a **Collection** or a **Stream** or an **Image**, it is common to want to perform some operation across some subset of that resource without worrying

about whether some other thread will change the resource halfway through your operation. In a situation like this, it's common to lock the resource, not **this.** we're of two minds on this: On the one hand, the principle of coupling leads to the thought that if the only thing that's vulnerable to being placed in an inconsistent state is the resource, then lock the resource, as locking **this** unnecessarily couples **this** and the resource. On the other hand, the principle of cohesion leads us to think that if one portion of the methods and resources in an instance are vulnerable to race conditions, but other methods and resources in the instance aren't, then maybe we ought to refactor. This is what we did with our **TwoPanel** class, splitting the initial class into two, and we think **Sharing3** is clearly a superior design to **Sharing2.**

Monitoring static value types

At this point, you shouldn't be surprised that this program quickly fails:

```
//:c16:RefCount1.cs
//How to synch a static value type? This program fails
using System;
using System.Threading;

class RefCount1 {
  static int refCount = 0;

  static RefCount1(){
    ThreadStart ts = new ThreadStart(ValCheck);
    Thread t = new Thread(ts);
    t.Start();
  }

  static void ValCheck(){
    Console.WriteLine("Starting ValCheck");
    while (true) {
      Console.WriteLine(DateTime.Now);
      if (refCount != 0 && refCount != 1) {
        Console.WriteLine(
          "Invalid: " + refCount);
        return;
      }
      Thread.Sleep(5000);
    }
  }
```

```
  RefCount1(){
    ThreadStart ts = new ThreadStart(Run);
    Thread t = new Thread(ts);
    t.IsBackground = true;
    t.Start();
  }

  //Thread unsafe
  void Run(){
    Console.WriteLine("Starting RefCount");
    while (true) {
      refCount++;
      refCount--;
    }
  }

  public static void Main(){
    for (int i = 0; i < 2; i++) {
      new RefCount1();
    }
  }
}///:~
```

The **Run( )** method, which is used by a bunch of threads, increments and then decrements **refCount**, while a thread running the static delegate method **ValCheck( )** stops the program if the value of **refCount** is ever not 0. When you run this, it is not long before **refCount** becomes either 2 or -1. This is what happens:

1. A **RefCount** instance thread increments **refCount**, making it 1.
2. That thread is interrupted by another thread, which increments **refCount** to 2.
3. The **ValCheck( )** thread interrupts *that* thread and sees a value of 2.

To synchronize access to the static value-type **refCount**, you have to wrap all access to the **refCount** object inside **lock** blocks that use some dummy object to synchronize on. You cannot **lock(refCount)** because it's a value type. And **lock(this)** is ineffective on static objects! So you have to create a static dummy object to serve as a guard for the critical sections:

```
static SomeObject myDummyObject = new SomeObject();
…
   static void ValCheck(){
      Console.WriteLine("Starting ValCheck");
      while (true) {
         Console.WriteLine(DateTime.Now);
            lock(myDummyObject){
               if (refCount != 0 && refCount != 1) {
                  Console.WriteLine(
                    "Invalid: " + refCount);
                  return;
               }
            }
         Thread.Sleep(5000);
      }
   }
…
void Run(){
      Console.WriteLine("Starting RefCount");
      while (true) {
         lock(myDummyObject){
            refCount++;
            refCount--;
          }
      }
   }
```

And indeed, that's the general solution to synchronizing static value types.

For the specific problem of reference counting, however, the **Interlocked** class allows you to increment, decrement, and assign to an **int** or **long** in a thread-safe and very, very fast manner. It can also compare two **int**s or two **long**s without requiring the overhead of the **Monitor** implementation.

Even aside from its synchronization capabilities, **Interlocked** should be used to generate serially increasing id values. Although

```
class StrangeButTrue{
    static int counter = 0;
    //Amazingly, this isn't thread-safe
    public static int Next(){
       return counter++;
```

```
    }
}
```

is fine for a single-threaded program, it's not thread-safe, since the ++ operator reads and assigns the value in two separate operations. A thread can come along after the read but before the assignment and therefore, in a multithreaded program, **Next( )** could return the same value twice.

The Monitor is not "stack-proof"

You might expect this program to freeze:

```
//:c16:RecursiveNotLocked.cs
using System;
using System.Threading;

class RecursiveNotLocked {
  int MAX_CALLS = 4;

  void RecursiveCall(int callCount){
    Console.WriteLine("Entering recursive call");
    lock(this){
      Console.WriteLine(
        "Exclusive access to this");
      if (callCount < MAX_CALLS) {
        RecursiveCall(++callCount);
      }
    }
  }

  public static void Main(){
    RecursiveNotLocked rl =
      new RecursiveNotLocked();
    rl.RecursiveCall(0);
  }
}///:~
```

The first time **RecursiveCall( )** is called, it acquires the **Monitor** to **this**. Then, from within that locked block, it calls **RecursiveCall( )**. What happens when it comes to the lock block on this second call? You might think "Well, it's a different call to the method, so it will block." But the right to enter the block is owned by the calling **Thread**; since it owns the lock on **this**, it continues into the **lock** block and recurses **MAX_CALLS** times.

Where to monitor

An object's thread-safety is entirely a function of the object's state. A class without any static or instance variables, consisting solely of methods that don't call methods that put it into **ThreadState.WaitSleepJoin**, is inherently thread-safe. Any variables or resources that affect whether your object is in a consistent state should be **private** or **protected** and only available to outside objects by way of properties or methods. It's just foolish to ever allow direct references to these critical resources from external objects. If instead you create properties and methods which consistently use the **Monitor** class (or the equivalent **lock** blocks), you'll save yourself considerable headaches when it comes to locating and debugging threading problems.

Cross-process synchronization with Mutex

The **Mutex** class is similar to the **Monitor** class, but is used to synchronize behavior across application domains. And instead of locking on an arbitrary object, the **Mutex** locks itself – owning the **Mutex** provides cross-process *mutual exclusion***.**

You attempt to acquire the **Mutex** by calling one of the overloaded **Mutex.WaitOne( )** methods. If you pass in no arguments, the calling thread goes into **ThreadState.WaitSleepJoin** until the **Mutex** is available. Or you can call **Mutex.WaitOne( )** with a timeout value and a **bool** that interacts with the **SynchronizedAttribute**. If you do not use the **[Synchronized( )]** attribute, the value you use for this is irrelevant.

This example allows only one copy of the application to run on the machine at a time. To run this, open *two* console windows and run the program in both simultaneously.

```
//:c16:MutexDemo.cs
//Demonstrates "Application Singleton"
using System;
using System.Threading;

class MutexDemo {
  public static void Main(){
    Mutex mutex = new Mutex(false, "MutexDemo");
    bool gotTheMutex = mutex.WaitOne(0, true);
    while (gotTheMutex == false) {
      Console.WriteLine(
        "Another app has the Mutex");
```

```
      gotTheMutex = mutex.WaitOne(3000, true);
    }
    Console.WriteLine(
      "This application has the mutex");
    Thread.Sleep(10000);
    Console.WriteLine("Okay, I'm done");
  }
}///:~
```

When run, the output of the *second* instance of the application will be:

```
Another app has the Mutex
Another app has the Mutex
This application has the mutex
```

(You may have one more or fewer "Another app has the Mutex" lines, depending on how fast you started the second application.)

The first line of **MutexDemo.Main( )** instantiates a .NET **Mutex** that corresponds to an OS-level mutex that is identified *at the OS-level* by the string "MutexDemo." The call to **WaitOne(0, true)** returns **true** immediately the first time the application is called – the OS-level "MutexDemo" mutex is available. Since **gotTheMutex** is **true**, the first application reports that it has the mutex and blocks for ten seconds.

If one or more additional applications are run within that ten second period, their calls to **WaitOne(0, true)** will immediately return **false** because the "MutexDemo" mutex is held by the first application. As long as **gotTheMutex** is **false**, the application reports it to the console and calls **WaitOne( )**, again, this time specifying a 3,000 millisecond timeout.

For the purposes of this book, the only thing that you can use a **Mutex** for is as the basis of cross-process blocking. If your work involves .NET Remoting, you'll learn how methods and objects can be invoked and moved across process boundaries and the **Mutex** will become the basis of synchronizing such behaviors.

Deadlocks

We've discussed race conditions as one of the challenges of multithreading programming. To fight race conditions, we introduced the various methods that put the current **Thread** into **ThreadState.WaitSleepJoin**. Unfortunately, this introduces a new type of problem: the *deadlock*.

A deadlock is a situation where **Thread a** requires a resource locked by **Thread b**, which **Thread b** will not release until it acquires a resource locked by **Thread a**. Both threads enter **WaitSleepJoin** and, at best, the program times out. Of course, if it's just two threads you have a good chance of finding the problem and debugging it; the real joy of deadlocks comes when it's **Thread a** depending on **Thread b** depending on **Thread c ...** depending on **Thread z** that depends on **Thread a**.

The classic deadlock exemplar is the Dining Philosophers problem. Five philosophers, or in this case language designers, are seated around a table. Each coder has a chopstick to his (or her) left and right.

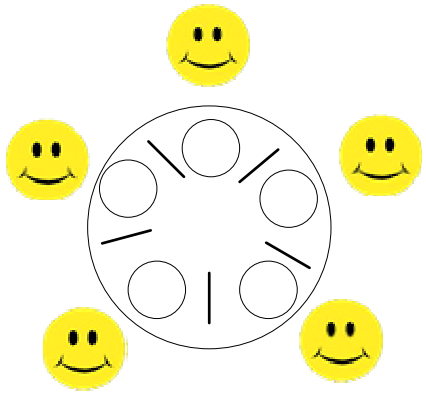


Figure 16-2: Language designers are happy when they have access to food

The coders do nothing but pontificate and eat. To eat, they pick up the chopstick on their left and then the chopstick on the right.

```
//:c16:DiningDesigners.cs
//Demonstrates dining philosophers problem
using System;
using System.Threading;

enum Desire {
  Pontificate, Eating
}

class Table {
  readonly int iDiners = 5;

  Diner[] diners;
```

```
  Chopstick[] chopsticks;

  Table(){
    diners = new Diner[iDiners];

    diners[0] = new Diner("Lovelace",
      "The difference engine rocks the most.");
    diners[1] = new Diner("Hejlsberg",
      "C# rules, ok?");
    diners[2] = new Diner("Stroustrup",
      "C++ is still the fastest");
    diners[3] = new Diner("Gosling",
      "Write once, run anywhere!");
    diners[4] = new Diner("van Rossum",
      "Brackets are bad");

    chopsticks = new Chopstick[iDiners];
    for (int i = 0; i < iDiners; i++) {
      Chopstick c = new Chopstick();
      chopsticks[i] = c;
      diners[i].LeftChopstick = c;
      diners[(i + 1) % iDiners].RightChopstick = c;
    }
  }

  public static void Main(){
    new Table();
  }
}

class Diner {
  readonly int basePonderTime = 10;
  readonly int ponderTimeRand = 0;
  readonly int baseEatTime = 10;
  readonly int eatTimeRand = 0;
  readonly int postEatTime = 10;
  readonly int betweenChopstickTime = 50;
  readonly int maxStartupDelay = 150;
  readonly int starvationTimeout = 100;

  static int totalMealsEaten = 0;
```

```
string name;
internal string Name{
  get{ return name;}
}

string opinion;

Thread myThread;

Chopstick left;
internal Chopstick LeftChopstick{
  set { left = value;}
  get{ return left;}
}
Chopstick right;
internal Chopstick RightChopstick{
  set{ right = value;}
  get{ return right;}
}

Desire desire = Desire.Pontificate;
static Random r = new Random();

internal Diner(string name, string opinion){
  this.name = name;
  this.opinion = opinion;
  ThreadStart ts = new ThreadStart(EatAndTalk);
  myThread = new Thread(ts);
  myThread.Start();
}

void EatAndTalk(){
  Thread.Sleep(r.Next(maxStartupDelay));
  while (true) {
    if (desire == Desire.Eating) {
      Eat();
    } else {
      Pontificate();
    }
  }
```

```
}

void Pontificate(){
  Console.WriteLine(name + ": " + opinion);
  Thread.Sleep(basePonderTime +
    r.Next(ponderTimeRand));
  desire = Desire.Eating;
}

void Eat(){
  GetChopsticks();
  Console.WriteLine(name + ": Eating");
  Thread.Sleep(baseEatTime  + r.Next(eatTimeRand));
  Console.WriteLine(name + ": Burp");
  Thread.Sleep(postEatTime);
  ReleaseChopsticks();
  Interlocked.Increment(ref totalMealsEaten);
  desire = Desire.Pontificate;
}

void GetChopsticks(){
  Console.WriteLine(name
    + ": Picking up left chopstick");
  left.Pickup(this);
  Thread.Sleep(betweenChopstickTime);
  Console.WriteLine(name
    + ": Picking up right chopstick");
  right.Pickup(this);
}

void ReleaseChopsticks(){
  left.Putdown();
  right.Putdown();
}

internal void Wait(int id){
  Console.WriteLine(name
    + ": Waiting for chopstick[{0}]", id);
  try {
    Thread.Sleep(starvationTimeout);
    Console.WriteLine(name
```

```
        + " starved to death waiting for "
        + " chopstick[{0}]", id);
      Console.WriteLine("Meals served: "
        + totalMealsEaten);
      myThread.Abort();
    } catch (ThreadInterruptedException) {
    }
  }

  internal void Interrupt(int id){
    myThread.Interrupt();
    Console.WriteLine(name
       + ": alerted to availability of "
       + "chopstick[{0}]", id);
  }
}

class Chopstick {
  static int counter = 0;
  int id = counter++;

  bool acquired = false;
  Diner waitingFor = null;

  internal void Pickup(Diner d){
    if (acquired == true) {
      waitingFor = d;
      //Must wait
      d.Wait(id);
    }
    acquired = true;
    Console.WriteLine(d.Name
      + ": Got chopstick[{0}]", id);
  }

  internal void Putdown(){
    Console.WriteLine(
      "Chopstick[{0}] released", id);
    acquired = false;
    if (waitingFor != null) {
      Console.WriteLine("Someone's waiting");
```

```
        waitingFor.Interrupt(id);
      }
      waitingFor = null;
    }
}///:~
```

When you run this program, the diners pontificate for a while and then start to eat. When a diner is in **Pickup( )** and they attempt to pick up a chopstick that is being used by their neighbor, they must wait. A reference to the **Diner** that is going to wait is stored in the **Chopstick**'s **waitingFor** variable. The call to **Diner.Wait( )** puts the diner's thread into **ThreadState.WaitSleepJoin**. If that **Thread** is not interrupted before the timeout, the diner starves to death. If, on the other hand, the current diner holding that chopstick calls **Putdown( )**, it calls **waitingFor.Interrupt( )** which in turn calls **Interrupt( )** on the thread that has been put into **WaitSleepJoin**.

When you run this program, it may run for a few seconds or it may run for quite a while, but eventually what will happen is that every diner will pick up the chopstick to their left and be waiting for the diner on their right to release their chopstick. All of the diners will starve to death.

There are several different ways to keep the diners from starving to death (for instance, if one of the group picks up the chopstick to their *right* first, no deadlock will occur), but the issue isn't really keeping dime-a-dozen language designers alive, it's developing good practices for minimizing deadlock.

One way to avoid a deadlock is to never wait for one resource while holding a lock on another. If there's no hold-and-wait, there can be no deadlock. However, theoretically this can trigger *livelock*, in which the threads release and then ask for the resources again with such perfect timing that they never resolve the issue. In Windows, this threat can be discounted; given the speed with which locks are acquired and the nature of the OS-level thread scheduler, livelock may not even be possible.

The most general way to beat deadlocks is to design the acquisition of locks such that there is no possibility of circular dependencies. Without a circular wait, deadlocks cannot occur. If you get into complex blocking code, use the overloads of **Monitor.Wait( )** and **Thread.Join( )** that take a maximum time before throwing **TimeoutException**s.

To minimize deadlock, you must work on the balance between properly controlling the critical regions in your code and the dependencies that arise from blocking. This is an area where commonly used programming language and the

.NET Framework fall short of what could be done. Multithreading is where memory management was a decade ago: a detail-oriented source of bugs that is the programmer's responsibility. Just as software and hardware advances made managed memory models practical on the desktop, so too can we hope that a more advanced parallel processing model will eventually be incorporated into the .NET Framework and into application programming languages.[2]

Object-Oriented Deadlock

Creating a safe multithreaded library is considerably more difficult than creating a single multithreaded application. Not only do you have to try to avoid deadlock in all the scenarios in which your library is used logically, you must never expose a virtual method that is called within a critical section. The problem is that if you declare a method as **virtual** and call it within a critical section (that is, a section in which you've acquired a **Monitor**), it is possible that the client programmer will override it in a way that creates a new thread and attempts to acquire the same **Monitor**. For this deadlock scenario to play out, the client must create a new thread since, as discussed on page 751, a call to acquire the **Monitor** within the same thread's calling stack will succeed. This is a subtle-enough requirement that this object-oriented deadlock can sneak by a lot of unit tests and code reviews.

The injunction goes against all virtual calls: those marked **virtual**, interfaces, abstract classes, and delegate calls are all vulnerable to this object-oriented deadlocking. In the following program, a **Library** object executes the method **Client.VirtualCall( )**. This works alright for **FineClient**, but **BadClient** deadlocks:

```
//:c16:VirtualCritical.cs
//Never expose a virtual method inside a lock
using System;
using System.Threading;
using System.Collections;

class Library {
```

[2] Cilk adds just 3 keywords to the C programming language, has a conceptually simple *work-stealing* scheduler, and yet is very efficient. Parallel processing *can* be made much more accessible to the average programmer; it just has not yet been a priority for language designers. Even without keywords, one can imagine using attributes to declaratively identify parallelization opportunities and constraints.

```
  Client client;
  public Client MyClient{
    set { client = value;}
  }

  Thread t;

  public void Run(){
    ThreadStart ts = new ThreadStart(ThreadCaller);
    t = new Thread(ts);
    t.Name = "Library thread";
    t.Start();
  }

  public void Stop(){
    t.Abort();
    t.Join();
  }

  void ThreadCaller(){
    while (true) {
      Console.WriteLine(Thread.CurrentThread.Name +
        " asking for lock");
      lock(this){
        Console.WriteLine(Thread.CurrentThread.Name +
          " acquired lock");
        client.VirtualCall();
        Thread.Sleep(1000);
      }
      Console.WriteLine(Thread.CurrentThread.Name +
        " released lock");
    }
  }
}

abstract class Client{
  public abstract void VirtualCall();

  Library l;
  public Library Library{
    set { l = value;}
```

```
  }

  protected bool callDone = false;

  public void LockAndTalk(){
    callDone = false;
    while (callDone == false) {
      Console.WriteLine(this.GetType() +
        " asking for lock");
      lock(l){
        Console.WriteLine(Thread.CurrentThread.Name +
          " acquired lock");
        Console.WriteLine("Virtual call executed");
        Thread.Sleep(1000);
        callDone = true;
      }
    }
    Console.WriteLine(Thread.CurrentThread.Name +
      " released lock");
  }
}

class FineClient: Client {
  public override void VirtualCall(){
    LockAndTalk();
  }
}

class BadClient: Client {
  public override void VirtualCall(){
    ThreadedLockAndTalk();
  }

  public void ThreadedLockAndTalk(){
    ThreadStart ts = new ThreadStart(LockAndTalk);
    Thread t = new Thread(ts);
    t.Name = "BadClient";
    t.IsBackground = true;
    t.Start();
    while (callDone == false) {
      Thread.Sleep(1000);
```

```
    }
  }
}

class TestingClass {
  Library l;
  TestingClass(Client c){
    Console.WriteLine("Testing " + c.GetType());
    l = new Library();
    l.MyClient = c;
    c.Library = l;
    l.Run();

    Thread.Sleep(10000);
    Console.WriteLine("Ending test now...");
    l.Stop();
  }

  public static void Main(){
    new TestingClass(new FineClient());
    new TestingClass(new BadClient());

  }
}///:~
```

The **Library** class contains a reference to a **Client** object, whose **VirtualCall( )** method is called within a **lock** block inside of **Library.ThreadCaller( )**. The two methods **Library.Run( )** and **Library.Stop( )** use previously discussed techniques to begin and end the **ThreadCaller( )** loop.

In addition to the **abstract** method **VirtualCall( )**, the **abstract** class **Client** specifies a method called **LockAndTalk( )**, which acquires a lock on the **Library** object, outputs something to the screen, waits a second, and then releases the lock. (This violates our preference to **lock(this)**, but it's the easiest code to demonstrate the danger of virtual method calls.)

FineClient just calls **LockAndTalk( )**. When **LockAndTalk( )** is called in **FineClient**, it is being executed in the same thread that executed **Library.ThreadCaller( )** and that owns the **Library** monitor. Therefore, **FineClient( )** works just fine.

BadClient( ) implements **VirtualCall( )** in a way that the **Library** author did not anticipate: it starts a new **Thread** whose **ThreadStart( )** delegate, a

method called **ThreadedLockAndTalk( )**, calls **LockAndTalk( )** from within a new thread. Notice that there is no explicit attempt on the part of the **BadClient** programmer to lock anything; the **BadClient** programmer is not doing anything obviously prone to failure. However, when **ThreadedLockAndTalk( )** calls **LockAndTalk( )** and that method attempts to acquire the lock on the **Library** object, the **lock** attempt is being executed from a *different* thread than the original thread in **Library.ThreadCaller( )**, which of course already has the lock on the **Library** object. The result is that although the **BadClient** programmer has done nothing obviously wrong, the **Library** deadlocks.

Writing a multithreaded library that is *reentrant*, that is, can be safely invoked from multiple threads, concurrently, and in a nested manner, is difficult enough at the best of times, but it is much harder if your library makes a virtual method call while holding onto a lock. Critical sections must be as controlled as possible; a virtual method call cedes that control and makes disaster all too likely.

Not advised: Suspend( ) **and** Resume( )

Referring back to Figure 16-1, you'll see that in addition to the **WaitSleepJoin** that we've discussed extensively, a call to **Thread.Suspend( )** will place a **Thread** into **ThreadState.SuspendRequested** and subsequently into **ThreadState.Suspended( )**. Unlike the static **Thread.Join( )** and **Thread.Sleep( )** methods, **Thread.Suspend( )** is an instance method. Thus, **Suspend** is useful in situations where, for some reason, the current **Thread** doesn't have sufficient knowledge to control its own scheduling.

If an inability to **lock(this)** is a "code smell" that suggests that a refactoring may be called for, a need to use **Thread.Suspend( )** is a stench. *Why* does the current thread not have enough knowledge to know what resources it needs to wait on or in what situations it should suspend processing? In a good object-oriented design, objects encapsulate both the state and behavior they need to fulfill their design contracts; in a good multithreaded design, the object that contains the **ThreadStart** delegate should take the responsibility to maintain the state and behavior necessary to properly control the **Thread** using the **ThreadStart** delegate.

Sometimes people try to use **Suspend( )** and **Resume( )** to prioritize the scheduling of their application's calculations, but that is precisely what the **Thread.Priority** property should be used for.

Threads and collections

The collection classes in .NET's **System.Collections** namespace are not thread-safe and behavior is "undefined" when collisions occur. This program illustrates the issue:

```
//:c16:SyncCol.cs
using System;
using System.Collections;
using System.Threading;

class SyncCol1 {
  public static void Main(){
    int iThreads = 25;
    SyncCol1 sc1 = new SyncCol1(iThreads);
  }

  int iThreads;
  SortedList myList;
  public SyncCol1(int iThreads){
    this.iThreads = iThreads;
    myList = new SortedList();
    TimedWrite(myList);
    myList =
    SortedList.Synchronized(new SortedList());
    TimedWrite(myList);
  }

  public void TimedWrite(SortedList myList){
    WriterThread.ExceptionCount = 0;
    WriterThread[] writerThreads =
    new WriterThread[iThreads];
    DateTime start = DateTime.Now;
    for (int i = 0; i < iThreads; i++) {
      writerThreads[i] =
        new WriterThread(myList, i);
      writerThreads[i].Start();
    }
    WaitForAllThreads(writerThreads);
    DateTime stop = DateTime.Now;
    TimeSpan elapsed = stop - start;
```

```
    Console.WriteLine(
      "Synchronized List: " + myList.IsSynchronized);
    Console.WriteLine(iThreads + " * 5000 = "
      + myList.Count + "? "
      + (myList.Count == (iThreads * 5000)));
    Console.WriteLine(
      "Number of exceptions thrown: " +
      WriterThread.ExceptionCount);
    Console.WriteLine(
      "Time of calculation = " + elapsed);
  }

  public void WaitForAllThreads(WriterThread[] ts){
    for (int i = 0; i < ts.Length; i++) {
      while (ts[i].Finished == false) {
        Thread.Sleep(1000);
      }
    }
  }
}

class WriterThread {
  static int iExceptionsThrown = 0;
  public static int ExceptionCount{
    get{ return iExceptionsThrown;}
    set{ iExceptionsThrown = value;}
  }

  Thread t;
  SortedList theList;

  public WriterThread(SortedList theList, int i){
    t = new Thread(new ThreadStart(WriteThread));
    t.IsBackground = true;
    t.Name = "Writer[" + i.ToString() + "]";
    this.theList = theList;
  }
  public bool Finished{
    get{ return isFinished;}
  }
  bool isFinished = false;
```

```
  public void WriteThread(){
    for (int loop = 0; loop < 5000; loop++) {
      String elName = t.Name + loop.ToString();
      try {
        theList.Add(elName, elName);
      } catch (Exception ) {
        ++iExceptionsThrown;
      }
    }
    isFinished = true;
    t.Abort();
  }
  public void Start(){
    t.Start();
  }
}///:~
```

The **Main( )** creates a **SyncCol1** class with a parameter indicating how many threads to simultaneously write to a collection. A **SortedList** is created and passed to the **TimedWrite** method. This method sets the static variable **ExceptionCount** of the **WriterThread** class to 0 and creates an array of **WriterThread**s. The **WriterThread** constructor takes the list and a variable. Each **WriterThread** creates a new thread, whose processing is delegated to the **WriterThread.WriteThread( )** method. The Is**Background** property of the **WriterThread**'s thread is set to true. Being able to create background "daemon" threads is very convenient, especially in a GUI, where the user can request a program closure at any time.

After the **WriterThread** constructor returns, the next line of **TimedWrite( )** calls the **Start( )** method, which in turn starts the inner thread, which in turn delegates processing to **WriteThread( ). WriteThread( )** loops 5,000 times, each time creating a new name (such as "Writer[12]237") and attempting to add that to **theList**. A **SortedArray** is backed by two stores – one to store the values and another to store a sorted list of keys (the keys may or may not be the same as the values).

The call to **Add( )** an element to the list is wrapped in a catch block. Since we are ignoring the details of the exception and only recording how many exceptions were thrown, the **catch** statement does not specify a variable name for the caught exception. Once the loop is finished, we set the **Finished** property of the **WriterThread,** kill the Thread, and return. Back in the **SyncCol1** class, the

main application thread goes through the array, checking to see if it's finished. If it's not, the main thread goes to sleep for 1,000 milliseconds before checking again. When all the **WriterThread**'s are **Finished**, the **WriteThread( )** writes some data on the experiment and returns.

After the initial call with a regular **SortedList**, we create a new **SortedList** and pass it to the static method **SortedList.Synchronized( )**. All the Collections have this static method, which creates a new, thread-safe Collection. To be clear, the program creates a total of 3 **SortedList**s: the one for the initial run through **WriteThread( )**, a second anonymous one, which is used as the parameter to **SortedList.Synchronize( )**, which returns a third one. After Chapter 12's description of the .NET I/O library, you should recognize the Decorator pattern in play.

When you run this program, you'll see that the first run with a plain **SortedList** throws a large number of exceptions (if you have a sufficiently speedy computer, you may get no exceptions, but if you increase the number of threads, eventually you'll run into trouble), while the list produced by **Synchronized( )** adds all the data flawlessly. You'll also see why Collections aren't synchronized by default: the thread-safe list takes something like 5-8 times the duration to complete.[3] You can find out if a Collection is synchronized or not by examining its **Synchronized** property, as **TimedWrite( )** does during its status-writing lines.

If, instead of using a list produced by **SortedList.Synchronized( )**, you put a **lock(theList)** block around the **Add( )** call, you'll get exception-less behavior on both runs as well. Curiously, if you do this, the synchronized list seems to always outperform the unsynchronized list by a small margin!

In general, though, the challenge of working with collections and threads is not the thread-safety of the underlying collection, but the inherent challenge of objects being added, deleted, or changed by threads while your current thread tries to deal with the collection as a single logical unit. For instance, in an object with an instance object called **myCollection**, whether the Collection is **Synchronized** or not, the lines

```
int i = myCollection.Count;
```

[3] If you said "But it's the same Big O!" give yourself a gold star. If you said, "But ignorant hacks would confuse library performance with language performance and compare thread-safe collections to non-thread-safe collections, and on the basis of simplistic benchmarks write that C# has a performance problem, just as they did with Java!" give yourself a platinum star.

```
Object o = myCollection[i];
```

are inherently thread-unsafe because another thread might have removed element **i** before the second line is executed. If the class obeys the recommendation that the only references to a critical resource like **myCollection** are internal, any method that accesses **myCollection** can simply **lock(this)** and achieve thread-safety.

The **foreach** keyword uses the **IEnumerator** interface to traverse a collection. You can also get an **IEnumerator** directly by calling **GetEnumerator( )** on any of the collection classes. The **IEnumerator** methods **MoveNext( )** and **Reset( )** will throw an **InvalidOperationException** if their originating collection is changed during the enumerator's lifetime. This is true even if the collection is synchronized – traversing a non-locked data structure is inherently thread-unsafe.

It's not always possible to design classes that don't expose internal instance or static collections, but give it a hard try before giving up on the attempt. Can you **Clone** the collection? Use the *Proxy* pattern to return, not the collection itself, but an interface to your own thread-safe methods? If not, be prepared for some long debugging sessions, because it's a good bet that any time you open the door to multithreading defects, someone will introduce them.

Summary

Multithreaded programming cannot be done casually. If you never **lock** anything, you will face race conditions. If you **lock** excessively, you will have deadlocks.

- To ensure state in a multithreaded environment, you must protect critical sections with **lock** blocks (or the equivalent calls to **Monitor.Enter( )** and, within a finally block, **Monitor.Exit( )**). A critical section may be as short as a single use of the ++ operator, so you must be painstaking in your consideration of all operations involving the critical state of your class.
- If non-**readonly** non-value data is ever shared between threads, every thread that writes *or reads* the data must obtain a lock on the data before writing or reading the data. The synchronization mechanism is required for reliable interthread communication as well as for mutual exclusion.
- Hold locks for as short a time as possible. The longer you hold a lock, the greater the chance for a deadlock.

- Don't call virtual methods from within a critical section. A virtual method can be implemented in any way and may cause a deadlock.
- Flaws in threading logic may very well escape all unit and acceptance testing, and are often difficult to recreate. Proper multithreading is the most demanding aspect of C#'s programming model.

Exercises

1. Starting with a single thread, write a program that counts as high as it can in 30 seconds, then doubles the number of threads used, counts for another 30 seconds, doubles the thread count, and so forth. Run it until things go awry. Graph the results.

2. Refactor any unresponsive user interfaces that you have created in previous exercises. Use threads to increase perceived appearance.

3. Write a splashscreen class for C# that, in addition to showing a graphic, shows a status message as different classes load and initialize. (Hint: Use the static constructor)

4. Write a custom control that draws an animated sine wave that appears to scroll like an oscilloscope

5. Write a custom control that slowly scrolls text in a **Label**.

6. Write a custom control that does "Star Wars"-style scrolling, with the text transformed so that it appears to recede into the distance as it approaches the top of the control.

7. Using the suggestion in the text (one language designer is contrarian and picks up the chopsticks in reverse order), save the dining language designers.

8. Using "dining philosophers" as your starting search term, use the Internet to research a different strategy for saving the dinner guests. Implement the solution.

9. Write a Mandelbrot or Julia set explorer. Use threads to maintain a responsive interface.

10. Using code from either the previous exercise or the wavelet transform code from earlier chapters, write a program that performs a number of such time-consuming computations in "batch mode." First run the program so that all the time-consuming computations are done with a

single thread, then with two threads sharing the load, then four, then eight, etc. Graph the results. If possible, repeat the experiment on a multiprocessor machine[4].

[4] Check both Microsoft and Intel Websites for information on the latest JIT compilers.

17: XML

Exensible Markup Language (XML) will power the second half of the Web Revolution. Everything that's happened with the Web to date, the rise of e-commerce, worldwide business and personal presence, and the shift in perception of computers from business devices to general purpose information appliances, has been built on the shaky foundation of HTML, a format specifically designed *not to do* many of the tasks for which it has been coopted. HTML is a brilliant format for creating and writing scientific papers; outside of that application, it succeeds despite itself.

XML is a text-based format for specifying data. Where HTML is best used for browser-rendered human consumption, XML is best used for program-to-program communication of structured data. Sometimes the XML consuming program will do little more than render the data for immediate human consumption à la a Web browser, but just as often, the consuming program will use the data for more complex behavior that may involve long-lasting internal state modifications or further XML-based conversations with the serving program.

XML may be generated in a variety of ways. Indeed, one of the great benefits of XML is that it allows a human with a text editor to stand in for an unavailable programmatic source or sink of the XML data; a Web Designer need not hit a live server in order to design an XSLT-based browser-based interface, a Web Services programmer need not fill out a form and click a button to generate a SOAP request.

For the purposes of learning about XML, this chapter will read and write XML files. The next chapter, 18, will have more to say about XML that is generated in different processes, either on the same machine or over the Web.

XML structure

An *XML Application* is a specification of the syntax and semantics of a data structure. Scalable Vector Graphics is an XML Application, the Schools Interoperability Framework is an XML Application, the Open Travel Alliance Specification is an XML Application, etc. This can be confusing: "Application" has come to be synonymous with "computer program," so "XML application" is heard by most people as "a computer program that uses XML." So people often refer to the XML-based specification as an *XML standard*, or an *XML format*, even though those terms, too, might lead to confusion.

So to develop anything meaningful in XML requires two steps: one, you must learn about XML itself, then, every time you tackle a new domain, you must learn about the XML application (aka specification, standard, format) of the domain. As a programmer, it's imperative that you learn XML itself, as a career strategy, you would be well advised to become highly cognizant of the specifications in vertical industries that appeal to you.

XML data structures are treelike; they consist of a *Root Element* that has some number of sub-*Elements* that, in turn, have some number (perhaps zero) of sub-Elements. All Elements have a Name and all Elements may contain zero-or-more *Attributes* (not to be confused with .NET Attributes). *Non-Empty Elements* contain one piece of text-based *Element Data*.

An XML *Document*, which is the high-level container that contains an instance of an XML data structure, must contain *well-formed* data – a data structure starting from the root element in which every element is closed; non-empty elements are closed by a tag matching the opening element but beginning with a slash (/), empty elements are closed by putting a slash at the end of the element.

An XML Document must contain an *XML Declaration* that asserts that what follows is XML and that specifies the encoding. In addition, an XML Document may contain *processing instructions*, which are instructions to the application that is processing the data. One common processing instruction is the **xml-stylesheet** instruction, which specifies the way the XML data should be transformed into a human-readable format. Figure 17-1 illustrates the structure of a simple XML document and how it might be visualized as a data structure.

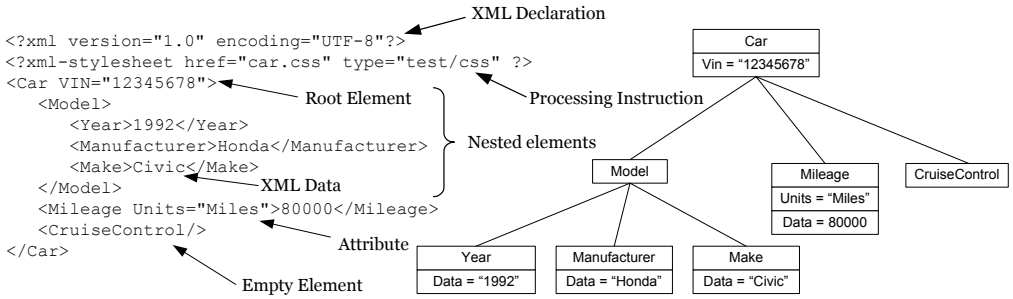


Figure 17-1: An XML document can be viewed as a tree-like data structure

The preceding description of XML should make it clear that XML follows neither the object-oriented model nor the relational model. XML is more hierarchical than either the object-oriented world of loosely interacting objects or the relational world of tightly related tables.[1] This emphasis on hierarchies, though, is the source of one very desirable trait of XML: An XML element can be viewed either as a complete data tree or it can be viewed as a stream of individual *nodes*.

XML as a stream

This example reads the XML document shown in Figure 17-1 from a file. As it's reading the file, it outputs some information on the nodes as it finds them.

```
//:c17:CarStreamReader.cs
//Reads an XML file
using System;
using System.IO;
using System.Xml;

class CarStreamReader {
  CarStreamReader(string fileName){
    XmlTextReader xIn = new XmlTextReader(fileName);
    try {
      while (xIn.Read()) {
        XmlNodeType typeOfNode = xIn.NodeType;
        Console.WriteLine("Read a {0} node",
          typeOfNode);
```

[1] Elliotte Rusty Harold has extended the classic "blind men and the elephant" fable to criticize OOP programmers, database programmers, Web developers, computer scientists, and technical writers who fail to see beyond their own biases when approaching XML.

```
        if (typeOfNode == XmlNodeType.Element) {
          Console.WriteLine("<{0}>", xIn.Name);
          if (xIn.HasAttributes) {
            while (xIn.MoveToNextAttribute()) {
              Console.WriteLine("[{0}] = {1}",
                xIn.Name, xIn.Value);
            }
          }
        }
        if (typeOfNode == XmlNodeType.Text) {
          Console.WriteLine("Text = " + xIn.Value);
        }
      }
    } finally {
      xIn.Close();
    }
  }
  public static void Main(){
    new CarStreamReader("car.xml");
  }
}///:~
```

First we specify that we'll be referencing the **System.Xml** namespace. In the **CarStreamReader( )** constructor, we instantiate an **XmlTextReader( )**. This class provides a forward-only, non-cached, stream-oriented reader of an XML source (in this case a file, but it can be any kind of **Stream**). Every time **XmlTextReader.Read( )** returns **true**, various properties in the **XmlTextReader** are set to reflect the value of the current XML node. The reason for this design is to increase performance: Because it does not create an object for each node, the **XmlTextReader** can rapidly read even very large XML streams.

After it's read the node, the **XmlTextReader.NodeType** property is one of the values of the **XmlNodeType** enumeration. This value is output to the console. If the node is an **XmlNodeType.Element** the value of **XmlTextReader.Name** is the element's name and **XmlTextReader.HasAttributes** specifies if there are attributes. If so, we loop with **XmlTextReader.MoveToNextAttributes( )**, which sets the **XmlTextReader**'s **Name** and **Value** properties appropriately.

If you run this program on the XML from Figure 17-1, you'll see that there are many nodes of type **XmlNodeType.Whitespace**. Often, you don't care about

such nodes; you can specify that the **XmlTextReader** ignore non-significant whitespace by switching its **WhitespaceHandling** property to **WhitespaceHandling.Significant**.

XML as a tree

Where **XmlTextReader** deals with XML in a stream-based manner, **XmlDocument** deals with XML as a treelike data structure. This program has a similar output to the previous, but because the **XmlDocument** is read entirely into memory before the data structure is traversed, it uses a recursive structure for traversal.

```
//:c17:CarDomReader.cs
//Loads an XML file
using System;
using System.IO;
using System.Xml;

class CarDomReader {
  CarDomReader(string fileName){
    XmlDocument doc = new XmlDocument();
    doc.Load(fileName);

    WriteInfo(doc);
  }

  void WriteInfo(XmlNode node){
    Console.WriteLine("Current node is of type {0}",
      node.NodeType);
    if (node.NodeType == XmlNodeType.Element) {
      Console.WriteLine("<{0}>", node.Name);
      foreach(XmlAttribute att
        in node.Attributes){
        Console.WriteLine("[{0}] = {1}",
          att.Name, att.Value);
      }
    }
    if (node.NodeType == XmlNodeType.Text) {
      Console.WriteLine("Text = " + node.Value);
    }
    foreach(XmlNode child in node.ChildNodes){
      WriteInfo(child);
```

```
    }
  }

  public static void Main(){
    new CarDomReader("car.xml");
  }
}///:~
```

After the **XmlDocument** object is created, **XmlDocument.Load( )** reads the entire contents of the passed-in **Stream** and constructs an in-memory representation of the XML. **XmlDocument** is itself just one type of **XmlNode**, so the call to **CarDomReader.WriteInfo( )** upcasts the **XmlDocument** to **XmlNode**. After printing out information about the **XmlNode**, **WriteInfo( )** iterates across each of the **XmlNode**s in the current node's **ChildNodes** collection. Each child node is passed as an argument to a recursive call of **WriteInfo( )**.

This example will throw an **XmlException** in **Load( )** if the document is poorly formed. The **CarStreamReader** example only throws an **XmlException** when **XmlTextReader.Read( )** attempts to read a poorly formed element. This is one difference in behavior that can help you choose between **XmlDocument** and **XmlTextReader**.

Writing XML

XML wouldn't be much good if you couldn't produce it programmatically. Just as with reading XML, you can choose between doing this in a forward-only, stream-based manner and doing it with the **XmlDocument** as an in-memory data structure. This example uses the latter to create the car document that we've been using:

```
//:c17:CarDomWriter.cs
//Creates an XML Document
using System;
using System.IO;
using System.Xml;
using System.Text;

class CarDomWriter {
  CarDomWriter(Stream outStr){
    XmlDocument doc = new XmlDocument();
    XmlDeclaration decl =
    doc.CreateXmlDeclaration("1.0", "UTF-8", null);
```

```
  doc.AppendChild(decl);
  string ssText =
    "type='text/xsl' href='book.xsl'";
  XmlProcessingInstruction pi =
  doc.CreateProcessingInstruction(
    "xml-stylesheet",ssText); doc.AppendChild(pi);
  XmlNode root = CreateCarNode(doc);
  doc.AppendChild(root);

  XmlTextWriter writer =
    new XmlTextWriter(outStr, new UTF8Encoding());
  writer.Formatting = Formatting.Indented;
  doc.WriteTo(writer);
  writer.Flush();
  writer.Close();
}

XmlNode CreateCarNode(XmlDocument doc){
  XmlElement car = doc.CreateElement("Car");
  car.SetAttribute("VIN","12345678");

  XmlNode model = CreateModelNode(doc);
  car.AppendChild(model);
  XmlNode miles = CreateMilesNode(doc);
  car.AppendChild(miles);
  XmlNode cruise = CreateCruiseNode(doc);
  car.AppendChild(cruise);
  return car;
}

XmlNode CreateModelNode(XmlDocument doc){
  XmlElement model = doc.CreateElement("Model");
  XmlElement year = doc.CreateElement("Year");
  year.InnerText = "1992";
  model.AppendChild(year);
  XmlElement mfr =
    doc.CreateElement("Manufacturer");
  mfr.InnerText = "Honda";
  model.AppendChild(mfr);
  XmlElement make = doc.CreateElement("Make");
  make.InnerText = "Civic";
```

```
    model.AppendChild(make);
    return model;
  }

  XmlNode CreateMilesNode(XmlDocument doc){
    XmlElement miles = doc.CreateElement("Mileage");
    miles.SetAttribute("Units", "Miles");
    miles.InnerText = "80000";
    return miles;
  }

  XmlNode CreateCruiseNode(XmlDocument doc){
    XmlElement cruise =
      doc.CreateElement("CruiseControl");
    return cruise;
  }

  public static void Main(){
    FileStream outStr = new FileStream(
      "car.xml",FileMode.Create);
    try {
      new CarDomWriter(outStr);
    } finally {
      outStr.Close();
    }
  }
}///:~
```

After initializing an **XmlDocument** object, the task is basically to create each node and append it to the **XmlDocument**. Because an **XmlDocument** has a particular encoding, nodes are not created directly with the **new** command, but rather the **XmlDocument** has a suite of **Create*Xxx*( )** methods. This is the *Factory Pattern*.

The **XmlDocument.Create*Xxx*( )** methods do not automatically place the just-created node in the data structure, they must be placed by calls to **XmlDocument.AppendChild( )**, **XmlDocument.InsertBefore( )**, or **XmlDocument.InsertAfter( )**.

After creating and adding the XML declaration and processing instruction, we call our own method **CarDomWriter.CreateCarNode( )** that in turn calls the other **CarDomWriter.Create*Xxx*( )** methods. Each goes through similar

steps: First, an **XmlElement** node is created using the **XmlDocument.CreateElement( )** factory method. Then, if the element has an attribute, we use **XmlElement.SetAttribute( )** to specify the name and value. If the element is a non-empty element, it's data is set with **XmlElement.InnerText**. This property is actually a shortcut; if we wanted to *really* duplicate the behavior of **XmlDocument.Load( )**, we'd create an **XmlText** node, set its **Value** property, and append it to the **XmlElement**. But using **InnerText** eliminates two lines of code per assignment and doesn't do any harm, so we used that.

XmlElement.AppendChild( ) is used to create the document's tree – we attach the nodes for the manufacturer, year, and make to the node for the model which we append to the node for the car itself. By the time **CreateCarNode( )** returns, it contains the whole tree of **XmlElement**s and we can append it to the **XmlDocument** node after the **XmlProcessingInstruction**.

We create an **XmlTextWriter** object with a sink of the **Stream** passed in as an argument to the **CarDomWriter( )** constructor and an encoding corresponding to the value in our **XmlDeclaration**. **XmlTextWriter.Formatting** defaults to **Formatting.None**, which outputs the XML as a single line. By setting it to **Formatting.Indented**, the output is spread across several lines and contains extra whitespace, but is much more readable.

XmlDocument.WriteTo( ) takes the **XmlTextWriter** as an argument. The call to **XmlTextWriter.Close( )** flushes the **XmlDocument** to the **XmlTextWriter's Stream**.

XmlDocument.WriteTo( ) is the complement of **XmlDocument.Load( )**, which as we discussed loads an entire document into memory. Not surprisingly, the **XmlTextWriter** has a set of methods that complements the forward-reading, stream-based **XmlTextReader**.**Read( )** method discussed in the **CarDomReader** example.

```
//:c17:CarStreamWriter.cs
//Creates an XML Document
using System;
using System.IO;
using System.Xml;
using System.Text;

class CarStreamWriter {
  CarStreamWriter(Stream outStr){
    XmlTextWriter writer =
```

```
    new XmlTextWriter(outStr, new UTF8Encoding());
    writer.Formatting = Formatting.Indented;
    writer.WriteStartDocument();
    writer.WriteProcessingInstruction
      ("xml-stylesheet","type='text/xsl' " +
       "href='book.xsl'");
    WriteCarNode(writer);
    writer.WriteEndDocument();
    writer.Close();
  }

  void WriteCarNode(XmlTextWriter writer){
    writer.WriteStartElement("Car");
    writer.WriteAttributeString("VIN", "12345678");

    WriteModelNode(writer);
    WriteMilesNode(writer);
    WriteCruiseNode(writer);

    writer.WriteEndElement(); //Car
  }

  void WriteModelNode(XmlTextWriter writer){
    writer.WriteStartElement("Model");

    writer.WriteStartElement("Year");
    writer.WriteString("1992");
    writer.WriteEndElement(); //Year

    writer.WriteStartElement("Manufacturer");
    writer.WriteString("Honda");
    writer.WriteEndElement(); //Mnfctr.

    writer.WriteStartElement("Make");
    writer.WriteString("Civic");
    writer.WriteEndElement(); //Make

    writer.WriteEndElement(); //Model
  }

  void WriteMilesNode(XmlTextWriter writer){
```

```
    writer.WriteStartElement("Mileage");
    writer.WriteAttributeString("Units", "Miles");
    writer.WriteString("80000");
    writer.WriteEndElement(); //Mileage
  }

  void WriteCruiseNode(XmlTextWriter writer){
    writer.WriteStartElement("CruiseControl");
    writer.WriteEndElement();
  }

  public static void Main(){
    FileStream outStr = new FileStream(
      "car.xml",FileMode.Create);
    try {
      new CarStreamWriter(outStr);
    } finally {
      outStr.Close();
    }
  }
}///:~
```

The major difference between **XmlDocument.WriteTo( )** and the stream-based approach of **XmlTextWriter** are the paired **XmlTextWriter.WriteStart*Xxx*( )** and **XmlTextWriter.EndStart*Xxx*( )** methods that need to be carefully matched. **XmlTextWriter** is sophisticated enough so that it properly writes empty and non-empty elements properly based on the placement of the **XmlTextWriter.WriteEndElement( )** method – the **CruiseControl** node is written as an empty node, not as a non-empty node with no data.

XML serialization

In the previous examples that used **XmlDocument** or **XmlTextReader** and **XmlTextWriter** to read and write XML, the code becomes very predictable very quickly – all nodes are created the same way, the hierarchy is built the same way, etc. As soon as you begin working in XML, you begin to think about ways to get rid of this boilerplate and directly serialize to and from XML. The **System.Xml.Serialization** namespace provides exactly this behavior – the ability to transform an XML document into a data structure composed not of **XmlNode**s but of domain and, from a set of domain objects, the ability to create a well-formed XML document.

XML serialization only serializes the public properties of an object (it will also serialize public fields, but you shouldn't *have* public fields). Additionally, XML serialization requires that there be a public, no-args constructor for the class. These constraints mean that XML serialization is not appropriate if your object's internal state is not entirely expressed with public properties. Not every type *should* fulfill these constraints; XML serialization is a great feature, but it shouldn't drive your design.

The **XmlSerializer** class itself is not serializable because an **XmlSerializer** can (and should) be cheaply recreated at runtime; what little advantage there might be in serializing an instance is overwhelmed by the temporal problems associated with object serialization (see Chapter 13). In this case, **XmlSerializer** is tightly bound to the set of public signatures of its target type; if the target's signature changes, all previously serialized **XmlSerializer**s would break.

XML serialization has a lot of flexibility, but to use it in its barest form is easy: Create an **XmlSerializer** for the class you wish to serialize, create an instance of that class, and use **XmlSerialize.Serialize( )** to write XML to a specified **Stream**:

```
//:c17:CarStructure1.cs
//Demonstrates XML Serialization
using System;
using System.Xml.Serialization;

public class Car {
  public Car(){
  }

  string vin;
  public string VIN{
    get { return vin;}
    set { vin = value;}
  }

  public static void Main(){
    Car c = new Car();
    c.VIN = "12345678";
    XmlSerializer carScribe =
      new XmlSerializer(typeof(Car));
    carScribe.Serialize(Console.Out, c);
  }
```

```
}///:~
```

In this example, the **Car** class has just one public property, the Vehicle Identification Number (**VIN**). The **XmlSerializer carScribe** object is specific to the **Car** type as shown in the **XmlSerializer( )** constructor. Once the **carScribe** object is created, we use it to write the specific **Car** we created (**c**) to the screen. The output of this program is:

```
<?xml version="1.0" encoding="IBM437"?>
<Car xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <VIN>12345678</VIN>
</Car>
```

This is just a little different than the XML we've been reading and writing. The important difference is the addition of attributes within the **Car** element that define XML *namespaces*. The other difference is that the VIN has been output as a child element of the **Car** element, rather than as an XML attribute, as we've been doing in our previous examples.

The **System.Xml.Serialization** namespace contains a large number of .NET Attributes to control XML output. To specify that the VIN be output as an XML attribute we associate the .NET Attribute **XmlAttributeAttribute** with the **VIN** property:

```
  [XmlAttribute]
  public string VIN{
     get { return vin; }
     set { vin = value; }
  }
```

leads to:

```
<?xml version="1.0" encoding="IBM437"?>
<Car xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
VIN="12345678" />
```

This example outputs an XML document very similar to the ones we were writing out by hand:

```
//:c17:CarStructure2.cs
//Demonstrates XML Serialization
using System;
using System.Xml.Serialization;
```

```
public class Car {
  public Car(){ }

  string vin;
  [XmlAttribute]
  public string VIN{
    get { return vin;}
    set { vin = value;}
  }

  Model model;
  public Model Model{
    get { return model;}
    set { model = value;}
  }

  Mileage miles;
  public Mileage Mileage{
    get { return miles;}
    set { miles = value;}
  }

  AirConditioning air;
  [XmlElement("AirConditioning",IsNullable=true)]
  public AirConditioning Air {
    get { return air;}
    set { air = value;}
  }

  public static void Main(){
    Car c = new Car();
    c.VIN = "12345678";
    c.Model = new Model(1992, "Honda", "Civic");
    c.Mileage = new Mileage("Miles", 80000);
    c.Air = new AirConditioning();
    XmlSerializer xs = new XmlSerializer(typeof(Car));
    xs.Serialize(Console.Out, c);
  }
}
```

```
public class Model {
  public Model(){
  }

  public Model(int yr, string mfr, string make){
    this.yr = yr;
    this.mfr = mfr;
    this.make = make;
  }

  int yr;
  public int Year{
    get { return yr;}
    set { yr = value;}
  }

  string mfr;
  [XmlElement("Manufacturer")]
  public string Maker{
    get { return mfr;}
    set { mfr = value;}
  }

  string make;
  public string Make{
    get { return make;}
    set { make = value;}
  }

  int horsepower = 100;
  [XmlIgnore]
  public int Horsepower{
    get { return horsepower;}
    set { horsepower = value;}
  }
}

public class Mileage {
  string units;
  [XmlAttribute("Units")]
  public string Units{
```

```
    get { return units;}
    set { units = value;}
  }

  int val;
  [XmlText]
  public int Quantity{
    get { return val;}
    set { val = value;}
  }

  public Mileage(){  }

  public Mileage(string units, int val){
    this.units = units;
    this.val = val;
  }
}


public class AirConditioning {
  public AirConditioning(){  }
}///:~
```

which produces the output:

```
<?xml version="1.0" encoding="IBM437"?>
<Car xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
VIN="12345678">
  <Model>
    <Year>1992</Year>
    <Manufacturer>Honda</Manufacturer>
    <Make>Civic</Make>
  </Model>
  <Mileage Units="Miles">80000</Mileage>
  <AirConditioning />
</Car>
```

In addition to **XmlAttributeAttribute**, this example uses the **XmlElementAttribute** so that the **Car.Air** property is output as an empty element named <**AirConditioning** />and the **Model.Maker** property is output as **<Manufacturer>.**

The **Model.Horsepower** property is associated with the **[XmlIgnore]** attribute, which specifies that the **XmlSerializer** not include the property in the XML document.

Deserializing XML

The **XmlSerializer.Deserialize( )** method converts an XML document into an object of the appropriate type:

```
//:c17:CarFromFile.cs
//Compile with:
//csc /reference:CarStructure2.exe CarFromFile.cs
//Demonstrates XML Deserialization
using System;
using System.IO;
using System.Xml.Serialization;

public class CarFromFile {
  public static void Main(){
    XmlSerializer xs = new XmlSerializer(typeof(Car));
    FileStream str =
    new FileStream("car.xml", FileMode.Open);
    Car c = (Car) xs.Deserialize(str);
    str.Close();
    Console.WriteLine("{0} {1} {2}, {3} {4}",
      c.Model.Year, c.Model.Maker, c.Model.Make,
      c.Mileage.Quantity, c.Mileage.Units);
  }
}///:~
```

This example requires a reference to the **Car** and other domain objects defined in the **CarStructure** example, but so long as the public properties of the object suffice to put it into a viable state, the **XmlSerializer** can both write and read objects to any kind of **Stream**.

Can't serialize cycles

This is a perfectly legitimate object-oriented relationship:



Figure 17-2: Cyclical relationships can be hard to express in XML

In this relationship, a **Yang** contains a reference to a **Yin** and a **Yin** to a **Yang**. It is possible for both objects to contain references to each other; the data structure may have a *cycle*.

XML does not use references to relate objects, it uses containment. One object contains another, which in turn contains its subobjects. There are no references native to XML. If the **XmlSerializer** detects a cycle, it throws an **InvalidOperationException**, as this example demonstrates:

```
//:c17:YinYang.cs
//Throws an exception due to cyclical reference
using System;
using System.Xml.Serialization;

public class Yin {
  Yang yang;
  public Yang Yang{
    get{ return yang;}
    set { yang = value;}
  }
  public Yin(){
  }
}

public class Yang {
  Yin yin;
  public Yin Yin{
    get{  return yin;}
    set { yin = value;}
  }
  public Yang(){
  }

  public static void Main(){
    Yin yin = new Yin();
    Yang yang = new Yang();
    //Set up cycle
    yin.Yang = yang;
    yang.Yin = yin;

    XmlSerializer xs = new XmlSerializer(typeof(Yin));
    //Throws InvalidOperationException
```

```
    xs.Serialize(Console.Out, yin);
  }
}///:~
```

If you wish to use XML to serialize an object structure that might contain cycles, you will have to create your own proxy for references. This will always require the use of unique text-based ids in lieu of references, the use of **[XmlIgnore]** and the dynamic "reattachment" of references based on the XML Ids.

Throughout this book, we've often used the phrase "This is an example of the *X* design pattern." Here, we have what seems to be the opposite case, a situation where we see a common problem (XML serialization of cyclical references) and can identify a path towards a general solution. There's a certain temptation to design something and present it as "the *Mock Reference* pattern" (or whatever). However, probably the most distinctive feature of the seminal books in the patterns movement (*Design Patterns* and *Pattern-Oriented Software Architecture*) is that they were based on software archaeology; patterns were *recognized in* existing, proven software solutions. *There are no .NET patterns* yet and very few XML patterns; there simply has not been enough time for a variety of design templates to prove themselves in the field.

Having said that, let's take a crack at a serializable Yin-Yang object structure:

```
//:c17:SerializedYinYang.cs
// Can serialize cycles
using System;
using System.IO;
using System.Collections;
using System.Xml.Serialization;

public class Yin {
  static Hashtable allYins = new Hashtable();
  public static Yin YinForId(Guid g){
    return(Yin) allYins[g];
  }

  Yang yang;

  public Yang Yang{
    get{ return yang;}
    set {
      yang = value;
    }
```

```
  }

  Guid guid = Guid.NewGuid();
  [XmlAttribute]
  public Guid Id{
    get { return guid;}
    set{
      lock(typeof(Yin)){
        allYins[guid] = null;
        allYins[value] = this;
        guid = value;
      }
    }
  }

  public Yin(){
    allYins[guid] = this;
  }
}

public class Yang {
  Yin yin;
  [XmlIgnore]
  public Yin Yin{
    get{  return yin;}
    set { yin = value;}
  }

  public Guid YinId{
    get { return yin.Id;}
    set {
      yin = Yin.YinForId(value);
      if (yin == null) {
        yin = new Yin();
        yin.Id = value;
      }
    }
  }


  Guid guid = Guid.NewGuid();
```

```
  [XmlAttribute]
  public Guid Id{
    get { return guid;}
    set { guid = value;}
  }
  public Yang(){
  }

  public static void Main(){
    Yin yin = new Yin();
    Yang yang = new Yang();
    yin.Yang = yang;
    yang.Yin = yin;

    XmlSerializer xs = new XmlSerializer(typeof(Yin));

    MemoryStream memStream = new MemoryStream();
    xs.Serialize(memStream, yin);

    memStream.Position = 0;
    StreamReader reader = new StreamReader(memStream);
    string xmlSerial = reader.ReadToEnd();
    Console.WriteLine(xmlSerial);

    Console.WriteLine("Creating new objects");
    memStream.Position = 0;
    Yin newYin = (Yin) xs.Deserialize(memStream);
    xs.Serialize(Console.Out, newYin);

    Yang newYang = newYin.Yang;
    Yin refToNewYin = newYang.Yin;
    if (refToNewYin == newYin) {
      Console.WriteLine("\nCycle re-established");
    }
    if (newYin == yin) {
      Console.WriteLine("Objects are the same");
    } else {
      Console.WriteLine("Objects are different");
    }
  }
}///:~
```

This program relies on the **Guid** structure, which is a "globally unique identifier" value; both classes have **Id** properties associated with the **XmlAttributeAttribute** that can serve to uniquely identify the objects over time.[2] The **Yin** class additionally has a **static Hashtable allYins** that returns the **Yin** for a particular **Guid**. The **Yin( )** constructor and the **Yin.Id.set** method update the **allYins** keys so that **allYins** and **YinForId( )** properly return the **Yin** for the particular **Guid.**

The **Yang** class property **Yin** is marked with **[XmlIgnore]** so that the **XmlSerializer** won't attempt to do a cycle. Instead, **Yang.YinId** is serialized. When **Yang.YinId.set** is called, the reference to **Yang.Yin** is reestablished by calling **Yin.YinForId( )**.

The **Yang.Main( )** method creates a **Yin** and a **Yang**, establishes their cyclical relationship**,** and serializes them to a **MemoryStream**. The **MemoryStream** is printed to the console, gets its **Position** reset, and is then passed to **XmlSerializer.Deserialize( )**, creating *new* **Yin** and **Yang** objects. Although **newYin** and **newYang** have the same **Id** values and the same cyclical relationships that the original **yin** and **yang** had, they are new objects, as Figure 17-3 illustrates.

[2] It's theoretically possible for GUIDs generated at different times to have the same value, but it's exceedingly rare. That's why GUIDs are only "globally" and not "universally" unique.

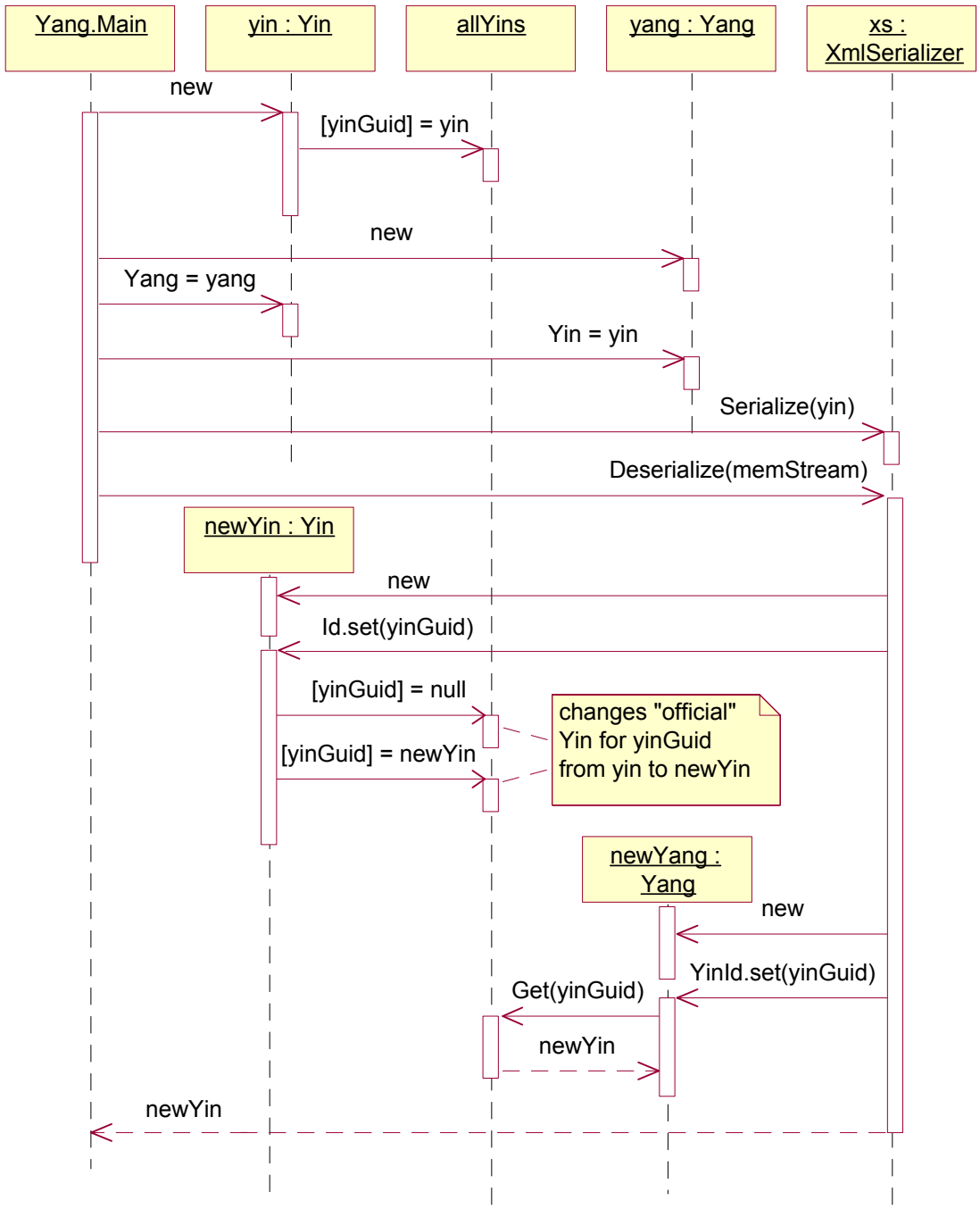

Yang.Main
yin : Yin
allYins
yang : Yang
xs : XmlSerializer
new
[yinGuid] = yin
new
Yang = yang
Yin = yin
Serialize(yin)
Deserialize(memStream)
newYin : Yin
new
Id.set(yinGuid)
[yinGuid] = null
[yinGuid] = newYin
changes "official" Yin for yinGuid from yin to newYin
newYang : Yang
new
YinId.set(yinGuid)
Get(yinGuid)
newYin
newYin

Figure 17-3: Reconstructing cycles from an XML document

The output of the program looks like this, although the **Guid**s will be different:

```
<?xml version="1.0"?>
<Yin xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
Id="8342aaa3-31e4-4d56-95fc-0959301a7ccf">
  <Yang Id="531ba739-673e-4840-a2c1-3027f9e60d9f">
    <YinId>8342aaa3-31e4-4d56-95fc-0959301a7ccf</YinId>
  </Yang>
</Yin>
Creating new objects
<?xml version="1.0" encoding="IBM437"?>
<Yin xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
Id="8342aaa3-31e4-4d56-95fc-0959301a7ccf">
  <Yang Id="531ba739-673e-4840-a2c1-3027f9e60d9f">
    <YinId>8342aaa3-31e4-4d56-95fc-0959301a7ccf</YinId>
  </Yang>
</Yin>
Cycle re-established
```

Schemas

So far, the only restriction that we've placed on the XML is that it be *well formed*. But XML has an additional capability to specify that only elements of certain types and with certain data be added to the document. Such documents not only are *well formed*, they are *valid*. XML documents can be validated in two ways, via *Document Type Definition* (DTD) files and via *W3C XML Schema* (XSD) files. The XML Schema definition is still quite new, but is significantly more powerful than DTDs. Most significantly, DTDs are not themselves XML documents, so you can't create, edit, and reason about DTDs with the same tools and code that you use to work with XML documents.

An XML Schema, on the other hand, is itself an XML document, so working with XML Schemas can be done with the same classes and methods that you use to work with any XML document. Another advantage of XML Schemas is that they provide for validity checking of the XML data; if the XML Schema specifies that an element must be a **positiveInteger**, then a variety of tools can validate the data flowing in or out of your program to confirm the element's values.

This XML Schema validates the data of the **CarStructure** example:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Car" nillable="true" type="Car" />
  <xs:complexType name="Car">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1"
        name="Model" type="Model" />
      <xs:element minOccurs="0" maxOccurs="1"
        name="Mileage" type="Mileage" />
      <xs:element minOccurs="1" maxOccurs="1"
      name="AirConditioning" nillable="true"
      type="AirConditioning" />
    </xs:sequence>
    <xs:attribute name="VIN" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="Model">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1"
        name="Year" type="xs:int" />
      <xs:element minOccurs="0" maxOccurs="1"
        name="Manufacturer" type="xs:string" />
      <xs:element minOccurs="0" maxOccurs="1"
        name="Make" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Mileage">
    <xs:simpleContent>
      <xs:extension base="xs:int">
        <xs:attribute name="Units" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="AirConditioning" />
  <xs:element name="Model" nillable="true"
    type="Model" />
  <xs:element name="Mileage" nillable="true"
    type="Mileage" />
  <xs:element name="AirConditioning" nillable="true"
    type="AirConditioning" />
</xs:schema>
```

We aren't going to go over the details, because you'll never have to handcraft an XML Schema until you're working at a quite advanced level. The .NET Framework SDK comes with a tool (**xsd.exe**) that can generate an XML Schema from an already compiled .NET assembly. This example was generated with this command line:

```
xsd CarStructure.exe
```

More commonly, you'll be working in a domain with a standards group that produces the XML Schema as one of its key technical tasks. If given an XML Schema, the **xsd** tool can generate classes with public properties that conform to the XML Schema's specification. In practice, this rarely works without modification of the schema; whether the fault lies in the **xsd** tool or the widespread lack of experience with XML Schema in vertical industries is difficult to say.

You can also use the **xsd** tool to generate a class descended from type **DataSet**. As discussed in Chapter 10, a **DataSet** is an in-memory, disconnected representation of relational data. So with the **xsd** tool, you can automatically bridge the three worlds of objects, XML, and relational data.

ADO and XML

From the discussion of ADO in Chapter 10, you'll remember that a **DataSet** is an in-memory representation of a relational model. An **XmlDataDocument** is an XML Doocument whose contents are synchronized with a **DataSet**, changes to the XML data are reflected in the **DataSet**'s data, changes to the **DataSet**'s data are reflected in the **XmlDataDocument** (as always, committing these changes to the database requires a call to **IDataAdapter.Update( )**).

This example quickly revisits the Northwind database and is essentially a rehash of our first ADO.NET program (you'll need a copy of **NWind.mdb** in the current directory):

```
//:c17:NwindXML.cs
//Demonstrates ADO to XML bridge
using System;
using System.Xml;
using System.Text;
using System.Data;
using System.Data.OleDb;

class NWindXML {
```

```
  public static void Main(string[] args){
    DataSet ds = Employees("Nwind.mdb");
    Console.WriteLine(
      "DS filled with {0} rows",
      ds.Tables[0].Rows.Count);
    //New lines begin here
    ds.WriteXml(Console.OpenStandardOutput());
    Console.WriteLine();
    ds.WriteXmlSchema(Console.OpenStandardOutput());
    //End new stuff
  }
  private static DataSet Employees(string fileName){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString =
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + fileName;
    DataSet ds = null;
    try {
      cnctn.Open();
      string selStr =
        "SELECT FirstName, LastName FROM EMPLOYEES";
      IDataAdapter adapter =
        new OleDbDataAdapter(selStr, cnctn);

      ds = new DataSet("Employees");
      adapter.Fill(ds);
    } finally {
      cnctn.Close();
    }
    return ds;
  }
}///:~
```

After retrieving a **DataSet** filled from the **Employees** table using the same ADO.NET code shown in Chapter 10,[3] we create a new **XmlDataDocument** that is linked to the **DataSet ds**. We then use **DataSet.WriteXml( )** and **DataSet.WriteXmlSchema( )** to present the XML view of the **DataSet** (an

[3] Well, almost the same. Instead of "SELECT * FROM EMPLOYEES" we limit the data to first and last names because the EMPLOYEES table contains photographs that make the returned XML unwieldy.

alternative would be to use an **XmlTextWriter).** When run, the output ends like this:

```
…
<Table>
    <FirstName>Anne</FirstName>
    <LastName>Dodsworth</LastName>
  </Table>
  <Table>
    <FirstName>Bob</FirstName>
    <LastName>Dobbs</LastName>
  </Table>
</Employees>
<?xml version="1.0"?>
<xs:schema id="Employees" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Employees" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Table">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="FirstName" type="xs:string"
minOccurs="0" />
              <xs:element name="LastName" type="xs:string"
minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A **DataSet** can also directly read an XML stream, either validating it against an existing schema or inferring a schema from the data. Here, the **car.xml** becomes the source of a **DataSet** and its inferred schema written to the screen:

```
//:c17:CarDataSet.cs
//Demonstrates DataSet from XML
using System;
```

```
using System.Xml;
using System.Text;
using System.IO;
using System.Data;
using System.Data.OleDb;

class CarDataSet {
  public static void Main(){
    DataSet carDataSet = new DataSet();
    carDataSet.ReadXml(
      "car.xml", XmlReadMode.InferSchema);

    Stream stdout = Console.OpenStandardOutput();
    carDataSet.WriteXmlSchema(stdout);
  }
}///:~
```

Combined with the **BindingContext** class discussed in Chapter 14, .NET's powerful infrastructure for XML, data, and display is beginning to fall into place. The final piece of the puzzle will have to wait until the next chapter, with our discussion of Web Services that use XML as the native "language" of Web-based conversations.

XPath navigation

While SQL is the native means of navigating relational data, XPath is the native means of navigating XML data. Since XML has a treelike structure, the natural way to get from one place to another is to specify a "path" up-or-down the branches from either the current Node or from the root Element. XPath is not itself expressed as an XML Document, rather it is reminiscent of file-system navigation commands.

The simplest XPaths are those that specify an absolute path to an object. Given this XML:

```
<!DOCTYPE set SYSTEM "docbookx.dtd">
<set>
  <book>
     <bookinfo>
         <title>Thinking in C#</title>
         <author>
               <personname>
                     <firstname>Larry</firstname>
```

```
                    <surname>O'Brien</surname>
                </personname>
            </author>
            <author>
                <personname>
                    <firstname>Bruce</firstname>
                    <surname>Eckel</surname>
                </personname>
            </author>
            <copyright>
                <year>2002</year>
                <holder>Larry O'Brien &amp;amp; Bruce
Eckel</holder>
            </copyright>
        </bookinfo>
        <preface>
            <title>Introduction</title>
        </preface>
        <chapter>
            <title>Those Who Can, Code</title>
        </chapter>
        <chapter>
            <title>Introduction To Objects</title>
        </chapter>
        <chapter>
            <title>Hello, Objects</title>
        </chapter>
    </book>
</set>
```

The XPath statement **/set/chapter/title** selects the titles of all the chapters. The command **//title** on the other hand, selects *all* title elements, including the title element that is contained below the **bookinfo** node.

The asterisk (*) command selects all elements contained within the specified path; **//bookinfo/*** selects the **title**, **author**, and **copyright** elements.

You move about an **XmlDocument** (or any other object that implements **IXPathNavigable**) via an **XPathNavigator**. This is the same philosophy that gives rise to the use of **IEnumerator**s on collection classes – separating the issues of traversal from the issues of the data structure. However, XPath takes this one step further: The **XPathNavigator** is responsible for selecting a

particular node, the **XPathNavigator** then produces an **XPathNodeIterator** to actually move about relative to the position selected by the **XPathNavigator.** These relationships are illustrated in Figure 17-4, which illustrates the behavior of this example:

```
//:c17:CarNavigator.cs
//Demonstrates XPathNavigator
using System;
using System.Xml;
using System.Xml.XPath;

class CarNavigator {
  CarNavigator(string fName){
    XmlDocument myDocument = new XmlDocument();
    myDocument.Load(fName);

    XPathNavigator myNavigator =
      myDocument.CreateNavigator();

    XPathNodeIterator myIterator =
      myNavigator.Select("//Model/*");

    while (myIterator.MoveNext()) {
      Console.WriteLine(
        "Node {0} Value {1}",
        myIterator.Current.Name,
        myIterator.Current.Value);
    }
  }

  public static void Main(){
    new CarNavigator("car.xml");
  }
}///:~
```

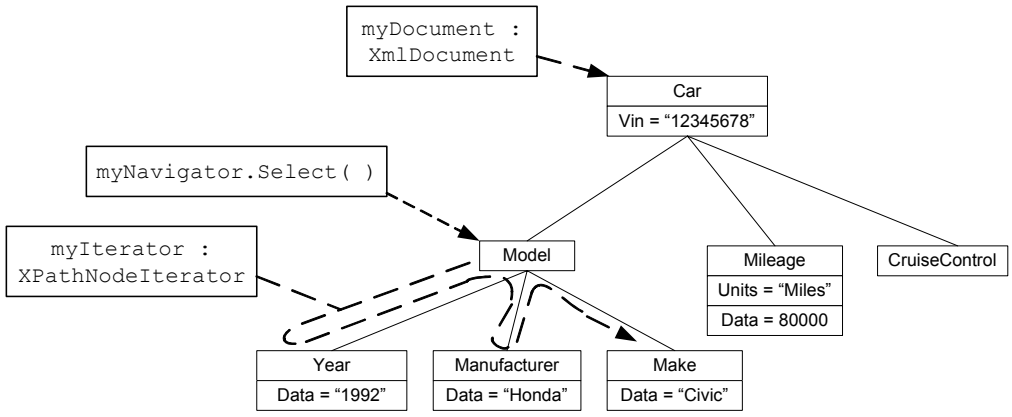


Figure 17-4: Navigating with XPathNodeIterator

The **CarNavigator( )** constructor loads the data structure from the **car.xml** file. **XmlDocument.CreateNavigator( )** is a factory method that generates the **XPathNavigator**. The **XPathNavigator.Select( )** is given an argument that translates as "select all the children nodes of all the **Model** nodes."

There's an overloaded version of **XmlDocument.CreateNavigator( )** that takes a reference to an **XmlNode** as an argument; it creates an **XPathNavigator** whose context node is not the root node (as was the case in the previous example) but the passed-in **XmlNode**. This allows you to work with subsets of a large XML document. The next example uses this method to create an **XPathNavigator** that navigates across a specific sale from the Northwind database. Additionally, the XPath selection statement in this example uses an argument to qualify the nodes returned by **XPath.Select( )**. When an XPath expression is qualified with square bracket notation, the contents of the square brackets are logically evaluated. In addition to logical expressions, 1-based index values can be used. Thus //chapter[3] or //chapter[title='Hello, Objects!'] both select the third chapter of this book's Docbook representation**.** In this example, this type of XPath qualifier is used to select only the sales of a particular employee:

```
//:c17:NWindNavigator.cs
//Demonstrates XPathNavigator subselection
using System;
using System.Xml;
using System.Xml.XPath;
using System.Data;
using System.Data.OleDb;
```

```
class NWindNavigator {
  public static void Main(string[] args){
    DataSet ds = EmployeesOrders("Nwind.mdb");
    Console.WriteLine(
      "DS filled with {0} rows",
      ds.Tables[0].Rows.Count);

    XmlDataDocument doc = new XmlDataDocument(ds);
    SelectSalesByLastName(doc, "Callahan");
  }

  private static void SelectSalesByLastName(
    XmlDataDocument doc, string lastName){
    XPathNavigator nav = doc.CreateNavigator();
    string xPathSel =
      "//Table[LastName='" + lastName + "']";
    XPathNodeIterator iter = nav.Select(xPathSel);
    while (iter.MoveNext()) {
      XPathNavigator saleNav = iter.Current.Clone();
      saleNav.MoveToFirstChild();
      string fName = saleNav.Value;
      saleNav.MoveToNext();
      string lName = saleNav.Value;
      saleNav.MoveToNext();
      string delDate = saleNav.Value;
      Console.WriteLine(
        "{0} {1} sold for delivery on {2}",
        fName, lName, delDate);
    }
  }

  private static DataSet EmployeesOrders(
    string fileName){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString =
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + fileName;
    DataSet ds = null;
    try {
      cnctn.Open();
      string selStr =
```

```
        "SELECT FirstName, LastName, OrderDate FROM"
        + " EMPLOYEES, ORDERS where "
        + " Employees.EmployeeId = Orders.EmployeeId";
      IDataAdapter adapter =
        new OleDbDataAdapter(selStr, cnctn);

      ds = new DataSet("Employees");
      adapter.Fill(ds);
    } finally {
      cnctn.Close();
    }

    return ds;
  }
}///:~
```

This example starts similarly to the previous Northwind examples, except instead of just loading data from the **Employees** table, this time the SQL **SELECT** statement joins the employees and their orders. The resulting dataset is approximately 800 lines long.

After the **DataSet** is returned to the **NWindNavigator( )** constructor, the constructor creates an **XmlDataDocument** synchronized with the **DataSet**. This **XmlDataDocument** and the last name of one salesperson become arguments to **NWindNavigator.SelectSalesByLastName( )**.

The **NWindNavigator.SelectSalesByLastName( )** method constructs an XPath selector of the form:

```
//Table[LastName='Callahan']
```

which creates an **XPathNodeIterator** for "Every node that is a Table and which in turn has a LastName child node whose value is 'Callahan.'"

We are not interested in the **Table** node, of course, we are interested in its children nodes: the **FirstName**, **LastName**, and **DeliveryDate** nodes. To navigate to them, we clone **XPathNodeIterator.Current** and call the resulting **XPathNavigator saleNav**. We use **XPathNavigator.MoveToFirstChild( )** and **XPathNavigator.MoveToNext( )** to traverse the instance of the **Table** node. We write to the screen the values of the various nodes in this sub-tree.

This combination of **XPathNavigator.Select( )**, **XPathNodeIterator.MoveNext( ),** and **XPathNavigator.MoveTo*Xxx*( )** methods is typical of use. The **XPathNavigator.MoveTo*Xxx*( )** methods are

significantly faster than **XPathNavigator.Select( )**, but not as flexible for complex navigation.

If you will be using a single **XPathNavigator.Select( )** statement repeatedly, you can use **XPathNavigator.Compile( )** to get a reference to an **XPathExpression**, which you can pass in to an overloaded version of **XPathNavigator.Select( )**.

XPath syntax has a few other tricks:

- The at symbol (@) is used to specify an attribute. Thus, you might select a specific car by using /Car[@VIN="12345678"] or a specific yin node with /yin[@Id="8342aaa3-31e4-4d56-95fc-0959301a7ccf">
- You can combine paths with the vertical bar operator (|) . //chapter | //preface selects all elements that are *either* chapters or prefaces .
- The *XPath axes* **child**, **descendant**, **parent**, **ancestor**, **following-sibling**, and **preceding-sibling** can be used to select nodes that have the appropriate structural relationship with the specified node. /set/book/chapter[1]/following-sibling selects the *second* chapter of the book (remember that XPath indices are 1-based, so chapter[1] selects the first chapter). The **child** axis is the default axis and need not be specified.

An XPath explorer

The best way to understand XPath is to experiment. This program loads XML documents into a **TreeView** control, and highlights element nodes corresponding to the XPath selection statement you type into the provided **TextBox**.

```
//:c17:XmlTreeView.cs
//Provides a graphical XPath navigator
using System;
using System.Text;
using System.Drawing;
using System.Xml;
using System.Xml.XPath;
using System.Windows.Forms;

class XmlTreeView : TreeView {
  XmlDocument doc;

  readonly Color DEFAULT_FORECOLOR = Color.Black;
```

```
readonly Color DEFAULT_BACKCOLOR = Color.White;
readonly Color HIGHLIGHT_FORECOLOR = Color.Blue;
readonly Color HIGHLIGHT_BACKCOLOR = Color.Red;

internal XmlTreeView(XmlDocument src){
  Init(src);
}

private void Init(XmlDocument src){
  doc = new XmlDocument();
  PopulateTreeFromNodeSet(
    src.CreateNavigator(), doc, this.Nodes);
}

void PopulateTreeFromNodeSet(
  XPathNavigator nav, XmlNode buildNode,
  TreeNodeCollection pNodeCol){
  do {
    string eType = nav.Name;
    string eVal = nav.Value;
    string nodeText = eType + ": " + eVal;
    XmlTreeNode node = null;
    switch (nav.NodeType) {
    case (XPathNodeType.Element) :
      node = new XmlTreeNode(eType, doc);
      break;
    case(XPathNodeType.Text) :
      node = new XmlTreeNode(eVal, doc);
      break;
    default:
      node = new XmlTreeNode(
        nav.NodeType.ToString(), doc);
      break;
    }
    pNodeCol.Add(node.TreeNode);
    buildNode.AppendChild(node);
    if (nav.HasChildren) {
      XPathNavigator clone = nav.Clone();
      clone.MoveToFirstChild();
      PopulateTreeFromNodeSet(
        clone, node, node.TreeNode.Nodes);
```

```
    }
  }while (nav.MoveToNext());
}

internal void Highlight(string xPath){
  ResetFormatting();
  try {
    XPathNavigator nav = doc.CreateNavigator();
    XPathNodeIterator iter = nav.Select(xPath);
    while (iter.MoveNext()) {
      XmlNode node =
        ((IHasXmlNode)iter.Current).GetNode();
      if (node is XmlTreeNode) {
        XmlTreeNode tNode = (XmlTreeNode) node;
        tNode.TreeNode.BackColor = Color.Red;
        tNode.TreeNode.ForeColor = Color.Blue;
        Invalidate();
      }
    }
  } catch (Exception e) {
    Console.WriteLine(e);
  }
}

internal void ResetFormatting(){
  foreach(TreeNode node in Nodes){
    ResetNodeFormatting(node);
  }
  Invalidate();
}

private void ResetNodeFormatting(TreeNode node){
  node.BackColor = DEFAULT_BACKCOLOR;
  node.ForeColor = DEFAULT_FORECOLOR;
  foreach(TreeNode child in node.Nodes){
    ResetNodeFormatting(child);
  }
}

internal void LoadFile(string fName){
  try {
```

```
      XmlDocument doc = new XmlDocument();
      doc.Load(fName);
      Nodes.Clear();
      Init(doc);
    } catch (Exception e) {
      Console.WriteLine(e);
    }
  }
}

class XmlTreeNode : XmlElement {
  internal XmlTreeNode(string val, XmlDocument doc)
  : base("",val,"",doc){
    tn = new TreeNode(val);
  }

  TreeNode tn;
  internal TreeNode TreeNode{
    get { return tn;}
    set { tn = value;}
  }
}

class XmlTreeViewForm : Form {
  XmlTreeView tl;
  TextBox xPathText;

  public XmlTreeViewForm(){
    Text = "XPath Explorer";

    XmlDocument doc = new XmlDocument();
    doc.Load("car.xml");
    tl = new XmlTreeView(doc);
    tl.Dock = DockStyle.Fill;
    Controls.Add(tl);

    Panel cPanel = new Panel();
    cPanel.Dock = DockStyle.Top;
    cPanel.Height = 25;

    Button xPathSel = new Button();
```

```
    xPathSel.Text = "Highlight";
    xPathSel.Dock = DockStyle.Left;
    xPathSel.Click += new EventHandler(NewXPath);
    cPanel.Controls.Add(xPathSel);

    xPathText = new TextBox();
    xPathText.Dock = DockStyle.Left;
    xPathText.Width = 150;
    cPanel.Controls.Add(xPathText);

    Label lbl = new Label();
    lbl.Text = "XPath: ";
    lbl.Dock = DockStyle.Left;
    lbl.Width = 60;
    cPanel.Controls.Add(lbl);

    Controls.Add(cPanel);

    MainMenu mainMenu = new MainMenu();
    MenuItem fMenu = new MenuItem("&File");
    mainMenu.MenuItems.Add(fMenu);
    MenuItem open = new MenuItem("&Open");
    fMenu.MenuItems.Add(open);
    open.Click += new EventHandler(FileOpen);
    fMenu.MenuItems.Add(new MenuItem("-"));
    MenuItem exit = new MenuItem("E&xit");
    exit.Click += new EventHandler(AppClose);
    fMenu.MenuItems.Add(exit);

    Menu = mainMenu;
  }

  void NewXPath(object src, EventArgs e){
    string xPath = xPathText.Text;
    tl.Highlight(xPath);
  }

  void FileOpen(object src, EventArgs e){
    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Filter = "XML files (*.xml)|*.xml";
    ofd.FilterIndex = 1;
```

```
    DialogResult fChosen = ofd.ShowDialog();
    if (fChosen == DialogResult.OK) {
      string fName = ofd.FileName;
      tl.LoadFile(fName);
    }
  }

  void AppClose(object src, EventArgs e){
    Application.Exit();
  }

  public static void Main(){
    Application.Run(new XmlTreeViewForm());
  }
}///:~
```

The concept of this program is that a **TreeView** with its **TreeNode**s is conceptually similar to an **XmlDocument** with its **XmlNode**s. To bridge the gap between the two classes, we create **XmlTreeView**, a type of **TreeView** that contains a reference to an **XmlDocument**, and a bunch of **XmlTreeNode**s, a type of **XmlElement** that contains a reference to a **TreeNode**. An **XPathNavigator** selects the **XmlNode**s in the **XmlDocument**, selected **XmlNode**s that are of type **XmlTreeNode** change the appearance of their corresponding **TreeNode**:

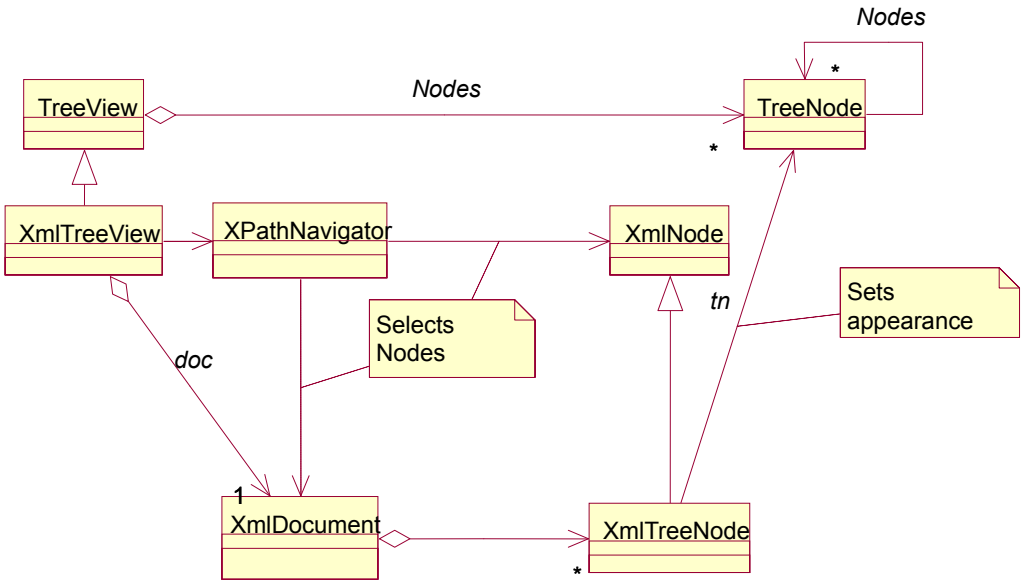


Figure 17-5: The class structure of XmlTreeView

The first task is populating a **TreeView** with the elements from an **XmlDocument**. **XmlTreeView.Init( )**, which the **XmlTreeView( )** constructor calls, creates a new **XmlDocument** called **doc**. The passed-in **XmlDocument src** is used as the basis for an **XPathNavigator** that will walk over all of its nodes, while the **XmlTreeView**'s inherited **TreeNodeCollection nodes** is the source of the **TreeNode**s. **Init( )** calls **XmlTreeView.PopulateNodeFromTreeSet** with hthese three arguments, **XPathNavigator nav**, **XmlNode doc** (upcast from **XmlDocument**), and **TreeNodeCollection Nodes**.

XmlTreeView.PopulateNodeFromTreeSet( ) iterates over each **XmlNode** in the **XPathNavigator nav** nodeset. A new **XmlTreeNode** is created for each node in the nodeset. The **XmlTreeNode( )** constructor in turn creates a new **TreeNode**. The **TreeNode** is appended to the passed-in **TreeNodeCollection** and the **XmlTreeNode** is appended to the **XmlDocument doc**. If the node has children, the method clones the **XPathNavigator** and recursively calls itself, this time with arguments set to the cloned **XPathNavigator** (pointing to a child node of the original **XmlDocument**), the just-created **XmlTreeNode,** and a reference to the just-created **TreeNode**. The end result is that

XmlDocument doc contains **XmlTreeNode**s, each of which maintains a reference to its **TreeNode.**

When someone calls **XmlTreeView.Highlight( )** with a **string** argument corresponding to an XPath selector, the first step is to reset the formatting of each **TreeNode** in the **XmlTreeView**. This is done with a recursive call to **XmlTreeView.ResetNodeFormatting( )**, which simply sets the colors of each **TreeNode** to their defaults. Once **XmlTreeView.Highlight( )** has reset the colors, it calls **XmlDocument.CreateNavigator( )** on the **XmlDocument doc**, which contains nothing but **XmlTreeNode**s. So the **XPathNodeIterator** created by **XPathNavigator.Select( )** traverses over a bunch of **XmlTreeNode**s. Each **XmlNode** returned is downcast to **XmlTreeNode** and the appearance of its corresponding **TreeNode** is changed to highlight the selection.

When run, the program provides an easy-to-use explorer of XPath functionality:

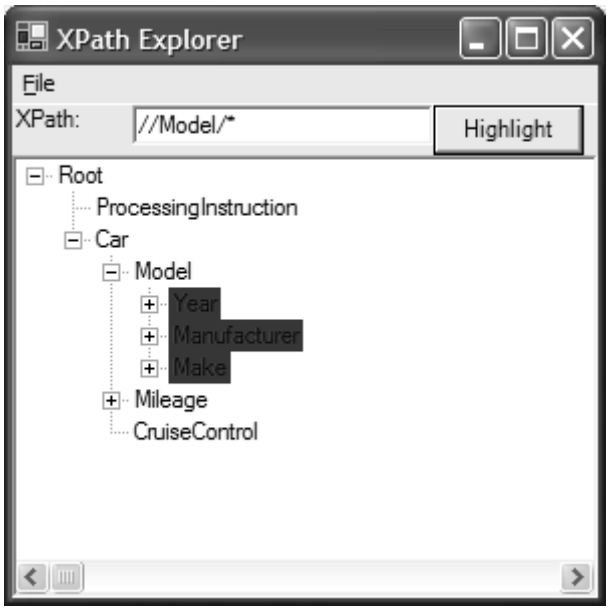


Figure 17-6: The XPath Explorer sample in action

Transforming a document

In addition to using XPath to select and navigate XML nodes, the .NET Framework provides the **System.Xml.Xsl** namespace[4] for transforming XML documents with Extensible Stylesheet Language Transformations (XSLT). XSLT is a declarative programming language that transforms XML documents from one form to another. The most common use of XSLT is to transform an XML document into a form suitable for display. Often, this means transforming it into XHTML, an XML-ified version of HTML. Blogs, for instance, typically use XML as their native storage format and use XSLT for the presentation.

The principle of separating domain data from presentation logic is one that we have praised in Chapter 14. Unfortunately, XSLT has not "broken out" as a mainstream technology for Web design; it is not supported in the major tools used by Web designers, the percentage of browsers incapable of displaying XSLT-based pages remains at least in the high teens, and the search engines have difficulty indexing XML-based sites. Although you can set things up so that the XSLT transformation occurs at the server if the client is using a non-compliant browser, the burden imposed can be significant, especially if you're really trying to exploit the power of XSLT. During the writing of this book, we changed the *www.ThinkingIn.Net* site from an XML and XSLT solution to straight HTML, primarily because of the search engine problem and the performance hit of server-side conversion.

Microsoft's decision to ship an XSLT class in the .NET Framework initially seems a little out of step with the lack of success of XSLT as a display technology, especially as Microsoft seems to be moving away from client-side XSLT towards Cascading Style Sheets (CSS) as the preferred mechanism for styling XML for browser display. However, the key is the deep relationship between **XmlDocument**s and **DataSet**s, especially in the **XmlDataDocument** class, which synchronizes changes between XML and relational data. Rather than viewing **XslTransform** just as a tool for creating interfaces, it is better to think of it as the final piece of the data-manipulation puzzle; you may be just as likely to use an **XslTransform** to restructure a database as to create a Web page.

In this example, we use a **DataRelation** to retrieve more than one table from the Northwind relational database.

```
//:c17:TwoTables.cs
```

---

[4] The namespace should be called **Xslt** since it does not support XSL Formatting Objects.

```
//Retrieves 2 tables from Northwind, outputs as XML
using System;
using System.IO;
using System.Xml;
using System.Xml.XPath;
using System.Data;
using System.Data.OleDb;

class TwoTables {
  public static void Main(string[] args){
    DataSet ds = EmpAndOrdRel("Nwind.mdb");

    FileStream outFile = new FileStream("EmpOrd.xml",
      FileMode.Create);
    ds.WriteXml(outFile);
    outFile.Close();
  }

  private static DataSet EmpAndOrdRel(
    string fileName){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString =
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + fileName;
    DataSet ds = null;
    try {
      cnctn.Open();
      string empStr =
        "SELECT EmployeeId, FirstName, LastName FROM "
        + "EMPLOYEES AS EMPLOYEES";
      OleDbDataAdapter empAdapter =
        new OleDbDataAdapter(empStr, cnctn);

      string ordStr =
        "SELECT * FROM ORDERS AS ORDERS;";
      OleDbDataAdapter ordAdapter =
        new OleDbDataAdapter(ordStr, cnctn);

      ds = new DataSet("TwoTables");
      empAdapter.Fill(ds,"Employees");
      ordAdapter.Fill(ds,"Orders");
```

```
      DataRelation rel = new DataRelation(
        "EmpOrder",
        ds.Tables["Employees"].Columns["EmployeeId"],
        ds.Tables["Orders"].Columns["EmployeeId"]);
      ds.Relations.Add(rel);
    } finally {
      cnctn.Close();
    }
    return ds;
  }
}///:~
```

The result is an XML document of this form:

```
<TwoTables>
  <Employees>
    <EmployeeId>1</EmployeeId>
    <FirstName>Nancy</FirstName>
    <LastName>Davolio</LastName>
  </Employees>
  <Employees>
    <EmployeeId>2</EmployeeId>
    <FirstName>Andrew</FirstName>
    <LastName>Fuller</LastName>
  </Employees>
…
  <Orders>
    <OrderID>10330</OrderID>
    <CustomerID>LILAS</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1994-11-16T00:00:00.0000000-
08:00</OrderDate>
    <RequiredDate>1994-12-14T00:00:00.0000000-
08:00</RequiredDate>
    <ShippedDate>1994-11-28T00:00:00.0000000-
08:00</ShippedDate>
    <ShipVia>1</ShipVia>
    <Freight>12.75</Freight>
    <ShipName>LILA-Supermercado</ShipName>
    <ShipAddress>Carrera 52 con Ave. BolÃ-var #65-98 Llano
Largo</ShipAddress>
```

```
    <ShipCity>Barquisimeto</ShipCity>
    <ShipRegion>Lara</ShipRegion>
    <ShipPostalCode>3508</ShipPostalCode>
    <ShipCountry>Venezuela</ShipCountry>
  </Orders>
…
</TwoTables>
```

Both **<Employees>** and **<Orders>** elements are placed as immediate children of the root node <**TwoTables**>. The following XSLT program is designed to transform the output of the **TwoTables** example:

```
<!-- Transforms EmpOrd.XML output from TwoTables.cs
-->
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/TwoTables">
   <EmployeeOrders>
      <xsl:apply-templates select="./Employees"/>
   </EmployeeOrders>
</xsl:template>

<xsl:template match="/TwoTables/Employees">
   <xsl:variable name="empId" select="./EmployeeId"/>
   <Employee>
   <xsl:attribute name="Id">
      <xsl:value-of select="$empId"/>
   </xsl:attribute>
      <FullName>
         <xsl:value-of select="./FirstName"/>
         <xsl:text> </xsl:text>
         <xsl:value-of select="./LastName"/>
       </FullName>
       <xsl:call-template name="OrdsForId">
          <xsl:with-param name="empId"
             select="$empId"/>
       </xsl:call-template>
   </Employee>
</xsl:template>

<xsl:template name="OrdsForId">
```

```
   <xsl:param name="empId"/>
   <xsl:apply-templates
     select="//Orders[EmployeeID=$empId]"/>
</xsl:template>

<xsl:template match="Orders">
   <xsl:variable name="Id">
      <xsl:value-of select="./OrderID"/>
   </xsl:variable>
   <Order>
   <xsl:attribute name="Id">
      <xsl:value-of select="$Id"/>
   </xsl:attribute>
   <DeliveryDate>
      <xsl:value-of select="./OrderDate"/>
   </DeliveryDate>
   </Order>
</xsl:template>

</xsl:stylesheet>
```

When applied to a target document, an XSL stylesheet creates a new document whose structure is determined by the stylesheet. An XSL stylesheet consists of a set of templates, which use either an XPath selector in a **match** attribute or a text **name** attribute as identifiers. Templates contain a mix of XSLT commands, such as **<xsl:call-template>** and **<xsl:for-each>**, and non-XSLT elements. The first template in this stylesheet matches the <**TwoTables>** root element and creates a new root element <**EmployeeOrders>**. The contents of the **<EmployeeOrders>** element are filled with the results of applying templates to the nodeset specified in the **<xsl:apply-templates select="./Employees">**. The second template matches each of these **<Employees>** nodes. The second template shows some additional capabilities of the XSLT language: variables, whitespace output, and the ability to explicitly call a template rather than rely on XPath selectors. The second template calls the **OrdsForId** template with the value of the **empId** variable that reflects the **<EmployeeId>** node in the original XML document. **OrdsForId** uses this value to create an XPath selector that retrieves the orders for the particular employee. The final template of the stylesheet creates an <**Order>** element with an **Id** attribute and a **<SaleDate>** element that corresponds to the original **<OrderDate>** element.

This program applies the stylesheet, saved in a file named **EmpOrd.xsl**, to the **EmpOrd.xml** data produced by the **TwoTables** program:

```
//:c17:QuickTransform.cs
//Demonstrates XslTransform class
using System;
using System.Xml;
using System.Xml.XPath;
using System.Xml.Xsl;

class QuickTransform {
  public static void Main(){
    XslTransform xslt = new XslTransform();
    xslt.Load("empord.xsl");
    XPathDocument doc =
      new XPathDocument("empord.xml");
    XmlTextWriter writer =
      new XmlTextWriter(Console.Out);
    writer.Formatting=Formatting.Indented;
    xslt.Transform(doc, null, writer);
  }
}///:~
```

QuickTransform creates an **XslTransform** object and loads the XSL stylesheet. An **XPathDocument** provides a fast, read-only **IXPathNavigable** object intended especially for XSLT processing. **XslTransform.Transform( )** works with any **IXPathNavigable** object (including **XmlDocument** and **XmlDataDocument**), but **XPathDocument** is preferred for high-speed transformation.

The second argument to **XslTransform.Transform( )** is an **XsltArgumentList** that allows you to extend the power of the **XslTransform** object; such extensions are beyond the range of this discussion.

When run, this program transforms the output of **TwoTables** into an XML document of the form:

```
<EmployeeOrders>
  <Employee Id="1">
    <FullName>Nancy Davolio</FullName>
    <Order Id="10340">
      <SaleDate>1994-11-29T00:00:00.0000000-
08:00</SaleDate>
    </Order>
    <Order Id="10258">
```

```
      <SaleDate>1994-08-17T00:00:00.0000000-
07:00</SaleDate>
    </Order>
…etc…
  </Employee>
</EmployeeOrders>
```

While you could achieve this type of output by manipulating an **XmlDocument** programmatically, once understood, XSLT makes XML transformations much easier than the equivalent direct programming tasks.

Summary

XML is a non-compact, text-based format for specifying structured data. Its strengths come from its readability, editability, and structure, which is neither relational nor object-oriented but close enough to bridge these worlds and flexible enough to be viewed as either a stream or as a treelike structure. XML itself is not all that interesting, but XML is the lingua franca for an entire generation of emerging data-interchange standards.

The .NET Framework allows you to work with XML as either a stream, using **XmlTextReader** and **XmlTextWriter**, or as a tree, using **XmlDocument**, which encapsulates the Core of W3C DOM Levels 1 and 2. Additionally, the .NET Framework exposes several pieces of core functionality in XML forms; objects can be serialized to and from XML, as can ADO.NET data.

The advantage of viewing an XML document as a tree is that you can apply various traversals and transformations to it, using the same principles that apply to all data structures (see Chapter 10). In particular, XPath is an XML specification that allows you to specify complex traversals that are themselves expressed in XML and XSLT allows you to specify complex transformations in XML.

The advantage of viewing an XML document as a stream is that it makes event-driven processing simple, and one can begin acting on XML data before (and whether or not) a complete document has been read. This is particularly valuable when moving XML to and from a network connection, but may also be the right approach any time XML documents will grow very large.

XML has emerged as the de facto standard for serializing data over the Web, but XML does not have a native way to express the full complexity of object-oriented data structures, such as reference cycles in which two objects refer to each other.

If these sorts of things are necessary to the application, they must be added programmatically.

Although XML is not a perfect solution, it has absolutely exploded onto the programming scene and will dominate over-the-net data transfer for the next decade at least. Because .NET embraces XML as the glue that binds stream, database, and object models together, it is important to develop a good understanding of XML's strengths and weaknesses.

Exercises

1. Investigate XML specifications in an industry or topic in which you are interested. Possibilities range from education (the Schools Interoperability Framework at *http://www.sifinfo.org/*) to travel (the Open Travel Alliance at *http://www.opentravel.org/*) to chemistry (Chemical Markup Language at *http://www.xml-cml.org/*).

2. Write a program that compares the time needed to count the number of nodes in a large XML document using an **XmlTextReader** (stream-based) with the time required by an **XmlDocument** (document-based) approach.

3. Modify the program in the previous example to select a random number of nodes (say, 100) and prints these nodes to the screen. Compare the times required by the stream-based versus document-based approach.

4. Write a program that creates an XML document that represents a deck of cards:

```
<Deck>
   <Card>
      <Suit>Spade</Suit>
      <Value>Ace</Value>
   </Card>
… etc …
</Deck>
```

5. Write a program to shuffle the "deck of cards" created in the previous example. This will require investigating the difficulties associating with creating a "perfect" shuffle algorithm.

6. Create an XML schema that describes the "deck of cards" from the previous examples.

7. Write a program that transforms the shuffled "deck of cards" XML document into a "deal" of four five-card hands:

```
<Deal>
   <Hand id="1">
      <Card>
         <Suit>Spade</Suit>
         <Value>Ace</Value>
      </Card>
      <Card>
         <Suit>Club</Suit>
         <Value>2</Value>
      </Card>
      … etc …
   </Hand>
   <Hand id="2">
      … etc…
   </Hand>
   … etc …
</Deal>
```

8. Repeat the above exercise but create the "Deal" solely by applying an XSLT transform to a "shuffled deck."

9. Write a program that ranks the five-card hands from the previous exercises according to standard poker rules.

10. Using what you have learned from the previous exercises, estimate the feasibility of creating an XML standard for the description of card games.

18: Web Programming

The Web has profound economic consequences for any industry that derives value from the flow of information, which is to say, all industries. As the desktop becomes a legacy form factor, cutting edge programming has moved in two directions: onto the server and into post-desktop devices such as handhelds, tablets, and phones. The .NET Framework makes programming for both these worlds straightforward. Programming non-desktop devices does not rely on radically different "enterprise" or "mobile" programming models or types. Rather, attributes, managed memory and threading, the scalable database architecture of ADO.NET, and the power of XML are added to the solid object-oriented support of the C# language.

Network programming in .NET is easy; a characteristic of networked .NET applications is how little code is devoted to network-specific issues. This trend is taken to its extreme with the **WebMethodAttribute**, which makes a method Web-callable with 11 keystrokes.[1] Although **WebMethodAttribute** is likely to be the most commonly used way you will expose methods to the Web, we're going to give a Cook's tour of more explicit ways to send data over the network.

Identifying a machine

Of course, in order to tell one machine from another to make sure that you are connected to the machine you want, there must be some way of quniquely identifying machines on a network. Early networks were satisified to provide

[1] And the purchase of a Microsoft server operating system!

unique machines within the local network. However, with IP (Internet Protocol) becoming by far the most common way of connecting computers, every machine *in the world* can have a unique identity. This is accomplished with the IP address, a 32-bit number that is expressed in the "dotted quad" format that has become so familiar.[2] Even more often, the DNS (Domain Name Service) is used to look up the particular IP address of a human-readable name such as **www.ThinkingIn.Net**.

.NET provides the **IPAddress** class to encapsulate the address. You can create an **IPAddress** for a specific 32-bit number, but you are far more likely to use **IPAddress.Parse( )** method to create an **IPAddress** from a dotted-quad **string** or the **Resolve( )** or **GetHostByName( )** methods of the **Dns** class.

The following program uses **Dns.GetHostByName( )** to produce your IP addresses. To use it effectively, you need to know the name of your computer. You can find this in the **Computer Name** tab in the **My Computer Properties** dialog.

```
//:c18:WhoAmI.cs
//Resolves current IP addresses
using System;
using System.Net;

class WhoAmI {
  public static void Main(string[] args){
    string machineName = args.Length == 0 ?
      "localhost" : args[0];
    IPHostEntry addresses =
    Dns.GetHostByName(machineName);
    foreach(IPAddress ip in addresses.AddressList){
      Console.WriteLine(ip);
    }
  }
}///:~
```

If you run this program with no command-line arguments, it will resolve the address of "localhost," which always resolves as 127.0.0.1. You can also use it find that *www.microsoft.com* resolves to several IP addresses.

---

[2] Curiously, .NET's **IPAddress** class uses a 64-bit **long** rather than a 32-bit **uint** to store this number. This is especially curious because IPv6, the long-awaited replacement of the 32-bit IPv4 is *128* bits in length, so it's not a case of the class having forward compatibility.

In addition to an address, IP uses a second more-specific location called a *port*, which is supposed to allow fine-tuning of what services are made available to the outside world. Of course, most system administrators are paranoid about opening ports to the outside world, perhaps because they do not understand that a process must be *listening* at the port to introduce a vulnerability.[3] This has created the absurd situation where there are more services than ever, but the majority of them are "tunneled" through the Web Server at port 80, thus *reducing* the ability of system administrators to quickly gain a clear picture of what services are causing what activity on the network.

Sockets

Data sent over the Internet is split into *datagrams*, each of which contains a header containing addressing information and a payload containing data. The developers of Berkeley UNIX did a great service to the world by abstracting all the bookkeeping details associated with acknowledging, retrying, and reassembling all this data so that it appears as a stream no different than that which is read from a local file system. The facility for doing this is called *Berkeley sockets* and .NET exposes them via **Socket** and related classes in the **System.Net.Sockets** namespace.

In the .NET Framework, you use a socket to connect two machines, then you get a **NetworkStream** that lets you treat the socket like any other IO stream object. You get this **NetworkStream** from an even higher-level abstraction than **Socket**, though: a **TcpListener** that a server uses to listen for incoming connections, and a **TcpClient** that a client uses in order to initiate a conversation. Once a client makes a socket connection, the **TcpListener** returns (via the **AcceptTcpClient( )** method) a corresponding *server-side* **TcpClient** through which direct communication will take place. From then on, you have a **TcpClient** to **TcpClient** connection and you treat both ends the same. At this point, you use **TcpClient.GetStream( )** to produce the **NetworkStream** objects from each **TcpClient**. You will usually decorate these with buffers and formatting classes just like any other stream object described in Chapter 12.

Whois for ZoneAlarm

This example shows how easy it is to write a socket-based utility using the **TcpClient** class. Most people do not have a dedicated firewall machine at their

---

[3] Although even one open port makes the server or network visible and thus potentially a target. However, if you run a Web server, your potential enemies will know your IP address.

home. While Windows XP provides a personal firewall, for various reasons one of use (Larry) runs ZoneAlarm from Zone Labs. While the payware version of ZoneAlarm provides a **whois** program for determining information on specific IP addresses, it is a manual process. After seeing a surprisingly large volume of alerts logged; he wanted to see if any addresses in particular were causing trouble. This program extracts IP addresses from the ZoneAlarm logfile, counts the number of events from that IP and, if the number of events exceeds a threshold, runs a **whois** query on the IP address:

```
//:c18:ZALogAnalyzer.cs
//Parses ZoneAlarm logfiles for suspicious IP sources
//Compile with:
//csc /reference:Whois.exe ZALogAnalyzer.cs
using System;
using System.Collections;
using System.IO;
using System.Net;
using System.Net.Sockets;

class ZALogAnalyzer {
  readonly int MULTI_HITTER_THRESHOLD = 500;

  StreamReader GetFile() {
    StreamReader log =
    File.OpenText("ZALog.txt");
    return log;
  }

  void ReadFile() {
    Hashtable sources = new Hashtable();
    StreamReader log = GetFile();
    string line = null;
    int i = 0;
    while ((line = log.ReadLine()) != null) {
      BlockInfo info = BlockInfo.Build(line);
      if (info != null && info is FWin) {
        IPAddress src = ((FWin)info).Origin;
        Object cnt = sources[src];
        if (cnt == null)
          sources[src] = 1;
        else {
```

```
            sources[src] = ((int)cnt) + 1;
          }
          i++;
          if ((i % 1000) == 0) {
            Console.WriteLine(i);
          }
        }
      }
      log.Close();
      Console.WriteLine(i);
      Console.WriteLine(sources.Count);
      int multiHitters = 0;
      foreach(IPAddress src in sources.Keys){
        if ((int)sources[src] >
            MULTI_HITTER_THRESHOLD) {
          multiHitters++;
          Console.WriteLine(
            src + ": " + sources[src]);
          Whois.OwnerOf(src);
        }
      }
      Console.WriteLine(
        "{0} multihitters,", multiHitters);
    }

    public static void Main() {
      ZALogAnalyzer prog = new ZALogAnalyzer();
      prog.ReadFile();
    }
}

class BlockInfo {
    private const int I_FWIN = 14;

    //Can't be instantiated by others
    protected BlockInfo(){}

    public static BlockInfo Build(string csv) {
      string[] components =
      csv.Split(new Char[]{',',':', '(', ')'});
      switch (components.Length) {
```

```
    case I_FWIN :
      return new FWin(components);
    default:
      return null;
    }
  }
}

class FWin : BlockInfo {
  //Protect construction
  protected FWin(){}

  DateTime time;
  IPAddress originator;
  public IPAddress Origin{
    get{return originator;}
    set{ originator = value;}
  }
  int targetPort;
  bool wasTCP;
  string flags;

  internal FWin(string[] comp) {
    string dateTime =
    string.Format("{1} {2}:{3}", comp);
    time = DateTime.Parse(dateTime);
    originator = IPAddress.Parse(comp[6]);
    targetPort = Int32.Parse(comp[9]);
    wasTCP = (comp[10] == "TCP") ? true : false;
    flags = comp[12];
  }
}///:~ (Example continues with Whois.cs)
```

The **ZALogAnalyzer** starts by hard-coding a few assumptions: that only those IP addresses that generated more than 500 logged events are worthy of investigation, and that the ZoneAlarm log is in its default path. **ZALogAnalyzer.GetFile( )** uses **File.OpenText( )**, which returns a **StreamReader** for the ZoneAlarm log.

ZALogAnalyzer.ReadFile( ) loops over every line in the log. Each line is passed to the static method **BlockInfo.Build( )**. Data in the ZoneAlarm log is of this form (new lines are indicated with '\n') :

```
FWOUT,2002/05/09,01:30:07 -7:00
GMT,192.168.1.1:1026,1.2.3.4:53,UDP \n
FWIN,2002/05/09,01:57:22 -7:00
GMT,1.2.3.4:0,192.168.1.1:0,ICMP (type:3/subtype:13) \n
```

The first line indicates that ZoneAlarm blocked an outbound DNS query from the local machine (IP address 192.168.1.1) to the machine at IP address 1.2.3.4. For the purposes of *this* utility, we're going to ignore outbound events[4] and look only at inbound events. **String.Split( )** separates each line by the passed-in **Char** parameters. This will split the lines we're interested in into a **string[]** array of a characteristic **I_FWIN** length.

BlockInfo.Build( ) is a skeleton of a more capable method. It is defined as returning a **BlockInfo**. **FWin**, the only type returned by the current implementation, is a subtype of **BlockInfo**. However, different utilities might be interested in creating different types to interpret different line types, and one can imagine **BlockInfo.Build( )** recognizing them, putting them in their own **case**s, and returning different subtypes of **BlockInfo**. This is a variation on the *Builder* creational pattern; *Builder* uses a **Director** class that is not part of the inheritance hierarchy to determine the specific subtype to be created.

The **FWin( )** constructor uses **String.Format( )** to reassemble three components from the logfile line into a **string** appropriate for consumption by **DateTime.Parse( )**. The fourth line of the **FWin( )** constructor introduces our first network-specific code when **IPAddress.Parse( )** is used to transform the dotted-quad into an IP address.

Back in **ZALogAnalyzer.ReadFile( )**, if the results of parsing the logfile line result in the creation of a **FWin** object, the **Hashtable sources** is checked to see if it has the **IPAddress** as an existing key. If not, the **IPAddress** is added as a new key and the value is set to 1. If the **IPAddress** is an existing key, the value is cast to an **int** and incremented.

Since the ZoneAlarm logs can be quite large, every 1000 lines, we write an informational message to the console. Once the log is finished, we write the total number of lines and the number of different IP addresses associated with inbound events. Those **IPAddress**es associated with more than **MULTI_HITTER_THRESHOLD** events are sent to that static method **Whois.OwnerOf( )**.

---

[4] A similar utility that checks the target of outbound events might also be helpful in assuring that your machine has not been compromised.

```
//:c18:Whois.cs
//Barebones "whois" utility
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;

public class Whois {
  private static string
  RESOLVER = "whois.arin.net";
  private static readonly int WHOIS_PORT = 43;

  public static string OwnerOf(IPAddress address){
    string whoisquery = "whois " + address;
    StringBuilder response = new StringBuilder();
    TcpClient cxn = null;
    try {
      cxn = new TcpClient(RESOLVER, WHOIS_PORT);
      BufferedStream stream =
        new BufferedStream(cxn.GetStream());
      StreamWriter upStream =
        new StreamWriter(stream);
      upStream.WriteLine(whoisquery);
      upStream.Flush();
      StreamReader downStream =
        new StreamReader(stream);
      string aLine;
      do {
        aLine = downStream.ReadLine();
        response.Append(aLine + "\n");
      }while (aLine != null);
      Console.WriteLine(
        response.ToString());
    } finally {
      if (cxn != null)
        cxn.Close();
    }
    return response.ToString();
  }
```

```
  public static void Main(string[] args){
    IPHostEntry he = Dns.Resolve(args[0]);
    if (args.Length == 2) {
      RESOLVER = args[1];
    }
    if (he.AddressList != null &&
        he.AddressList.Length > 0) {
      IPAddress tgt = he.AddressList[0];
      Whois.OwnerOf(tgt);
    }
  }
}///:~
```

Whois functionality requires a service provider, by default this utility uses one from the American Registry for Internet Numbers although you can pass in an alternate provider on the command-line.

The **TcpClient** is the primary class for reading socket-based data. After declaring a **TcpClient cxn,** its used inside a **try** block so that we can know that we will always have an opportunity to close it. The **TcpClient( )** constructor takes a **string** and an **int** that is the whois port on the **RESOLVER**. **TcpClient.GetStream( )** returns a network connection as a **Stream** which can be used in any way discussed in Chapter 10. The **NetworkStream** returned can be used for both sending *and* receiving data over the network, which is exactly what is done here. The **StreamWriter upStream** is used to send the whois query to the whois provider, while the **StreamReader downStream** loops over the response, adding the lines to the **StringBuilder response**. When done, the result is printed on the console and the **finally** block ensures that the **TcpClient** is closed.

Whois.cs has a **Main( )** method to make it a more general-purpose utility than just as a tool for analyzing ZoneAlarm logs. The **Dns.Resolve( )** method accepts either a dotted-quad or human-readable Internet address. If you pass *two* arguments on the command-line, the second overrides the ARIN whois server.

Even though **Whois.exe** is a stand-alone program, you can still reference it from **ZALogAnalyzer** as if it were a library assembly. To compile **ZALogAnalyzer.cs**, first compile **Whois** and then compile **ZALogAnalyzer**:

```
csc /reference:Whois.exe ZALogAnalyzer.cs
```

Receiving incoming connections

The previous example used an existing whois server. If you wish to create your own server, you will use the **TcpListener** class. This program demonstrates a very simple server. All the server does is wait for a connection, then uses the **TcpClient** produced by that connection to create a **NetworkStream**. After that, everything it reads coming in it echoes back down until it receives the line END, at which time it closes the connection.

The client makes the connection to the server, then creates a **NetworkStream**. Lines of text are sent through the **Stream** and received lines are printed to the console (in this case, what is printed is just the echo of the lines sent up).

Here is the server:

```
//:c18:JabberServer.cs
// Very simple server that just
// echoes whatever the client sends.
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;

public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static readonly int PORT = 1711;
  public static void Main(){
    TcpListener server = new TcpListener(PORT);
    TcpClient cnxn = null;
    try {
      server.Start();
      Console.WriteLine("Started: " + server);
      // Blocks until a connection occurs:
      cnxn = server.AcceptTcpClient();
      Console.WriteLine(
        "Connection accepted: "+ cnxn);C#
      StreamReader reader =
        new StreamReader(cnxn.GetStream());
      StreamWriter writer =
        new StreamWriter(cnxn.GetStream());
      writer.AutoFlush = true;
      Console.WriteLine("Beginning receive loop");
```

```
      while (true) {
        String str = reader.ReadLine();
        if (str == "END") break;
        Console.WriteLine("Echoing: " + str);
        writer.WriteLine(str);
      }
      //Close connection, stop server
    } finally {
      try {
        Console.WriteLine("closing...");
        cnxn.Close();
      } finally {
        server.Stop();
      }
    }
  }
}///:~
```

You can see that the **TcpListener** just needs a port number, not an IP address (since it's running on *this* machine!). You call **TcpListener.Start( )** and **TcpListener.Stop( )** to control when you want to listen at the underlying port. After you have called **TcpListener.Start( )**, the call to **TcpListener.AcceptTcpClient( )**, *blocks* until some client tries to connect to it. That is, it's there waiting for a connection, but other processes can run (see Chapter 16). When a connection is made, **TcpListener.AcceptTcpClient( )** returns with a **TcpClient** object representing that connection.

The responsibility for cleaning up the sockets is crafted carefully here. Nested **try-finally** blocks are used to ensure that we will both close the **TcpClient cnxn** and stop the **TcpListener**. If the **ServerSocket** constructor fails, the program just quits (notice we must assume that the constructor for **ServerSocket** doesn't leave any open network sockets lying around if it fails). For this case, **main( ) throws IOException** so a **try** block is not necessary. If the **ServerSocket** constructor is successful then all other method calls must be guarded in a **try-finally** block to ensure that, no matter how the block is left, the **ServerSocket** is properly closed.

The same logic is used for the **Socket** returned by **accept( )**. If **accept( )** fails, then we must assume that the **Socket** doesn't exist or hold any resources, so it doesn't need to be cleaned up. If it's successful, however, the following statements must be in a **try-finally** block so that if they fail the **Socket** will still be cleaned up. Care is required here because sockets use important nonmemory

resources, so you must be diligent in order to clean them up (see Chapters 5 and 11).

The next part of the program looks just like opening files for reading and writing except that the **StreamReader** and **StreamWriter** are created from the **NetworkStream** object returned by **TcpClient.GetStream( )**. The **StreamWriter** object has its **Autoflush** property set to **true** so that after every call to **StreamWriter.Write( )** or **StreamWriter.WriteLine( )**, the buffer is sent over the network. Flushing is important for this particular example because the client and server each wait for a line from the other party before proceeding. If flushing doesn't occur, the information will not be put onto the network until the buffer is full, which in this example doesn't happen, resulting in the programs just sitting after the initial connection is made.

When writing network programs you need to be careful about using automatic flushing. Every time you flush the buffer a packet must be created and sent. In this case, that's exactly what we want, since if the packet containing the line isn't sent then the handshaking back and forth between server and client will stop. Put another way, the end of a line is the end of a message. But in many cases, messages aren't delimited by lines so it's much more efficient to not use auto flushing and instead let the built-in buffering decide when to build and send a packet. This way, larger packets can be sent and the process will be faster.

The infinite **while** loop reads lines from the **StreamReader reader** and writes information to the console and to the **StreamWriter writer**. Note that **reader** and **writer** could be any streams, they just happen to be connected to the network.

When the client sends the line consisting of "END," the program breaks out of the loop, closes the **TcpClient,** and stops the **TcpListener**.

Here's the client:

```
//:c18:JabberClient.cs
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;

public class JabberClient {
```

```
  private static readonly int PORT = 1711;
  public static void Main(string[] args){
    IPAddress addr = IPAddress.Loopback;
    if (args.Length == 1) {
      addr = Dns.Resolve(args[0]).AddressList[0];
    }
    Console.WriteLine("addr = " + addr.ToString());
    TcpClient client = new TcpClient();
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      Console.WriteLine("Client = " + client);
      client.Connect(addr, PORT);
      StreamReader reader =
        new StreamReader(client.GetStream());
      StreamWriter writer =
        new StreamWriter(client.GetStream());
      writer.AutoFlush = true;
      for (int i = 0; i < 10; i ++) {
        writer.WriteLine("howdy " + i);
        string str = reader.ReadLine();
        Console.WriteLine(str);
      }
      writer.WriteLine("END");
    } finally {
      Console.WriteLine("closing...");
      client.Close();
    }
  }
} ///:~
```

In **Main( )** you can see an alternative to **Dns.Resolve( )** for getting the local **IPAddress**; if this program is run without a command-line argument, the **IPAddress addr** is set to the special "loopback" IP address of 127.0.0.1.

Once the **TcpClient** object has been connected to the **JabberServer** with the call to **TcpClient.Connect( )**, the process of turning its **NetworkStream** into a **StreamReader** and **StreamWriter** is the same as in the server (again, in both cases you start with a **TcpClient**). Here, the client initiates the conversation by sending the string "howdy" followed by a number. Note that the buffer must again be flushed (which happens automatically since the **writer**'s **Autoflush**

property has been set to **true**). Each line that is sent back from the server is written to the console to verify that everything is working correctly. To terminate the conversation, the agreed-upon "END" is sent. If the client simply hangs up, then the server throws an exception.

You can see that the same care is taken here to ensure that the network resources represented by the **TcpClient** are properly cleaned up, using a **try-finally** block.

Sockets produce a "dedicated" connection that persists until it is explicitly disconnected. (The dedicated connection can still be disconnected unexplicitly if one side, or an intermediary link, of the connection crashes.) This means the two parties are locked in communication and the connection is constantly open. This seems like a logical approach to networking, but it puts an extra load on the network. Later in this chapter you'll see a different approach to networking, in which the connections are only temporary.

Serving multiple clients

The **JabberServer** works, but it can handle only one client at a time. In a typical server, you'll want to be able to deal with many clients at once. The answer is multithreading, and in languages that don't directly support multithreading this means all sorts of complications. In Chapter 16 you saw that multithreading in .NET is about as simple as possible, considering that multithreading is a rather complex topic. Because threading in C# is reasonably straightforward, making a server that handles multiple clients is relatively easy.

The basic scheme is to make a single **ServerSocket** in the server and call **accept( )** to wait for a new connection. When **accept( )** returns, you take the resulting **Socket** and use it to create a new thread whose job is to serve that particular client. Then you call **accept( )** again to wait for a new client.

In the following server code, you can see that it looks similar to the **JabberServer.java** example except that all of the operations to serve a particular client have been moved inside a separate thread class:

```
//:c18:MultiJabberServer.cs
// A server that uses multithreading to handle
//any number of clients
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Threading;
```

```
public class MultiJabberServer {
  // Choose a port outside of the range 1-1024:
  public static readonly int PORT = 1711;
  public static void Main(){
    new MultiJabberServer();
  }

  public MultiJabberServer(){
    TcpListener server = new TcpListener(PORT);
    try {
      server.Start();
      Console.WriteLine("Started: " + server);
      while (true) {
        // Blocks until a connection occurs:
        TcpClient cnxn = server.AcceptTcpClient();
        Console.WriteLine(
          "Connection accepted: "+ cnxn);
        new ServeOneJabber(cnxn);
      }
    } catch (Exception ex) {
      Console.WriteLine(ex);
    } finally {
      server.Stop();
    }
  }
}

class ServeOneJabber {
  TcpClient cnxn;

  internal ServeOneJabber(TcpClient cnxn){
    this.cnxn = cnxn;

    ThreadStart oneServer =
      new ThreadStart(Run);
    Thread svrThread = new Thread(oneServer);
    svrThread.Start();
  }

  public void Run(){
```

```
    try {
      StreamReader reader =
        new StreamReader(cnxn.GetStream());
      StreamWriter writer =
        new StreamWriter(cnxn.GetStream());
      writer.AutoFlush = true;
      Console.WriteLine("Beginning receive loop");
      while (true) {
        String str = reader.ReadLine();
        if (str == "END") break;
        Console.WriteLine("Echoing: " + str);
        writer.WriteLine(str);
      }
    } finally {
      cnxn.Close();
    }
  }
}///:~
```

The **MultiJabberServer( )** constructor starts the **TcpListener** and waits for a connection to be made. When the connection is made, it creates a **ServeOneJabber** object, passing it the **TcpClient** that represents the new connection.

The **ServeOneJabber( )** constructor stores the **TcpClient cnxn** in the instance variable of the same name, creates a **ThreadStart** delegate, and starts the new **Thread**. This constructor finishes quickly and allows **MultiJabberServer** to resume listening for other connection requests.

The **ServeOneJabber.Run( )** method is the thread delegate method and its content is identical to the original **JabberServer**'s logic of just echoing whatever is sent to it.

The responsibility for cleaning up the **TcpClient** can not be trusted to the **MultiJabberServer** that creates it, though. Rather, **ServeOneJabber.Run( )** takes on the responsibility of calling **TcpClient.Close( )** inside a **finally** clause.

Notice the simplicity of the **MultiJabberServer**. As before, a **TcpListener** is created and **AcceptTcpClient( )** is called to allow a new connection. But this time, the return value (a **TcpClient**) is passed to the constructor for **ServeOneJabber,** which creates a new thread to handle that connection. When the connection is terminated, the thread simply goes away.

To test that the server really does handle multiple clients, the following program creates many clients (using threads) that connect to the same server. The maximum number of threads allowed is determined by the **final int MAX_THREADS**. You can experiment with **MAX_THREADS** to see where your particular system begins to have trouble with too many connections.

```
//:c18:MultiJabberClient.cs
// Client that tests the MultiJabberServer by
//starting up multiple clients

using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Threading;

public class MultiJabberClient {
  private static readonly int PORT = 1711;
  private static readonly int MAX_THREADS = 25;
  public static void Main(string[] args){
    IPAddress addr = IPAddress.Loopback;
    if (args.Length == 1) {
      addr = Dns.Resolve(args[0]).AddressList[0];
    }
    Console.WriteLine("addr = " + addr.ToString());
    for (int i = 0; i < MAX_THREADS; i++) {
      new MultiJabberClient(i, addr);
    }
  }

  int index;
  IPAddress addr;

  MultiJabberClient(int index, IPAddress addr){
    this.index = index;
    this.addr = addr;
    ThreadStart del = new ThreadStart(Run);
    Thread clientThread = new Thread(del);
    clientThread.Start();
  }
```

```
  public void Run(){
    TcpClient client = new TcpClient();
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      Console.WriteLine("Client = " + index);
      client.Connect(addr, PORT);
      StreamReader reader =
        new StreamReader(client.GetStream());
      StreamWriter writer =
        new StreamWriter(client.GetStream());
      writer.AutoFlush = true;
      for (int i = 0; i < 10; i ++) {
        writer.WriteLine("howdy {0} from client {1} ",
          i, index);
        string str = reader.ReadLine();
        Console.WriteLine(str);
      }
      writer.WriteLine("END");
    } finally {
      Console.WriteLine("closing client {0}", index);
      client.Close();
    }
  }
} ///:~
```

The **MultiJabberClient** constructor takes an **int index** and **InetAddress**, stores them in local variables, and fires up a **Thread** using **MultiJabberClient.Run( )** for its behavior. **MultiJabberClient.Run( )** does just what **JabberClient** did: connect to the server with a **TcpClient**. You're probably starting to see the pattern: The **TcpClient** is always used to create some kind of **StreamReader** and/or **StreamWriter.** Here, messages are sent to the server and information from the server is echoed to the screen. However, the thread has a limited lifetime and eventually completes.

Communicating with Microsoft Messenger

The previous examples demonstrate how to connect two computers directly, but the past few years have seen an explosion in so-called "peer-to-peer" applications, which combine server-based directory and search capabilities with machine-to-machine direct transfers. This next example uses Microsoft Messenger to send pointless jabber to your friends and family.

This example differs from the others in this book by presenting an interface to an essentially undocumented object. Working with undocumented objects and APIs *is* part of the professional programmer's job. The associated business and technology risks must be addressed by management, but often the least-documented areas contain the greatest potential for creating innovative client value. This example also differs from the others in this book by presenting what is essentially a flat-out "hack" – directly modifying the Intermediate Language (IL) code of a generated class to make up for a defect in one of Microsoft's tools.

There is no .NET Framework managed class for communicating with Microsoft Messenger, so developing for Microsoft Messenger requires COM Interop, newsgroup archives, and trial-and-error. The first step for interacting with Microsoft Messenger is to use **tlbimp** to create a COM Interop wrapper:

```
tlbimp "c:\program files\messenger\msmsgs.exe"
```

You'll need to add a reference to the resulting **Messenger.dll** and mark your code with the **[STAThread]** attribute, which is always necessary when using COM Interop (chapter 14).

```
//:c18:MessengerSend.cs
//Command-line Microsoft Messenger sender
//Compile with:
//csc /reference:Messenger.dll MessengerSend.cs
using System;
using System.Text;
using System.Runtime.InteropServices;
using Messenger;

class MessengerSend {
  static readonly string USER="larry@ThinkingIn.Net";
  static readonly string PASS="NotMyRealPassWord";

  [STAThread] public static void Main(string[] args){
    if (args.Length < 2) {
      Console.WriteLine(
        "USAGE: MessengerSend recipient message");
      return;
    }
    MessengerSend ms = new MessengerSend();
    ms.Login(USER, PASS);
    IMsgrUser recipient =
    ms.FindByFriendlyName(args[0]);
```

```
  StringBuilder msg = new StringBuilder();
  for (int i = 1; i < args.Length; i++) {
    msg.Append(args[i] + " ");
  }
  if (recipient != null) {
    ms.SendMessage(recipient, msg.ToString());
  }
}

MsgrObjectClass msgrObj = new MsgrObjectClass();

void Login(string user, string pass){
  IMsgrServices svcs = msgrObj.Services;
  IMsgrService msn = svcs.PrimaryService;
  try {
    msgrObj.Logon(user, pass, msn);
  } catch (COMException logonException) {
    if (logonException.Message ==
      "Exception from HRESULT: 0x81000304.") {
      Console.WriteLine("Already logged in");
    } else {
      throw logonException;
    }
  }
}

IMsgrUser FindByFriendlyName(string fName){
  IMsgrUsers contacts =
    msgrObj.get_List(MLIST.MLIST_CONTACT);
  Console.WriteLine("I have {0} contacts",
    contacts.Count);
  for (int i = 0; i < contacts.Count; i++) {
    IMsgrUser contact = contacts.Item(i);
    if (contact.FriendlyName == fName) {
      return contact;
    }
  }
  return null;
}

void SendMessage(IMsgrUser recipient, string msg){
```

```
    IMsgrIMSession session =
      msgrObj.CreateIMSession(recipient);
    try {
      session.SendText(null, msg,
        MMSGTYPE.MMSGTYPE_ALL_RESULTS);
    } catch (Exception ex) {
      Console.WriteLine(ex);
    }
  }
}///:~
```

MessengerSend is hard-coded to login as a particular Passport user as specified by the **USER** and **PASS readonly** variables. If you don't have a Passport, you can get one for free from *http://www.passport.com*. The biggest practical challenge for programming with COM Interop is documentation; figuring out what objects to create and how to get from these objects the references to the interfaces you suspect you need and, once you have the interfaces, generating valid arguments and interpreting the results. If you use Visual Studio .NET, the Object Browser window is invaluable; Figure 18-1 shows an Object Browser view of **Messenger.DLL**. In the left-hand pane are the types exposed by the assembly; a large number of enumerations such as **MLOCALOPTION**, the **MsgrObject** interface, and the **MsgrObjectClass** class. After creating a managed DLL for COM Interop, you should look for the classes exposed – there will usually be only a handful of options and, naturally, these are the only objects that you can directly create (there will often be *many* interfaces, but as you know, interfaces cannot be directly instantiated). From there, it's just a matter of searching MSDN, newsgroup archives, and the Web in general to develop a route of inquiry. In this case, it turns out that the **MsgrObjectClass** is the root object from which we can send and receive information.

Natively, many COM calls are defined as returning a 32-bit integer known as an **HRESULT**. In Windows, the value of the **HRESULT** specifies either that the method succeeded or the particular reason why it failed. In .NET code, this is exactly the role of exceptions and with COM Interop, **HRESULT**s are automatically mapped into exceptions. Thus, **tlbimp** does a lot of the "heavy lifting" associated with exposing the COM server as an object-oriented library. As can be seen in the Object Browser figure, **tlbimp** automatically generates not just methods, but properties and events (we'll talk more about the events in the next example).

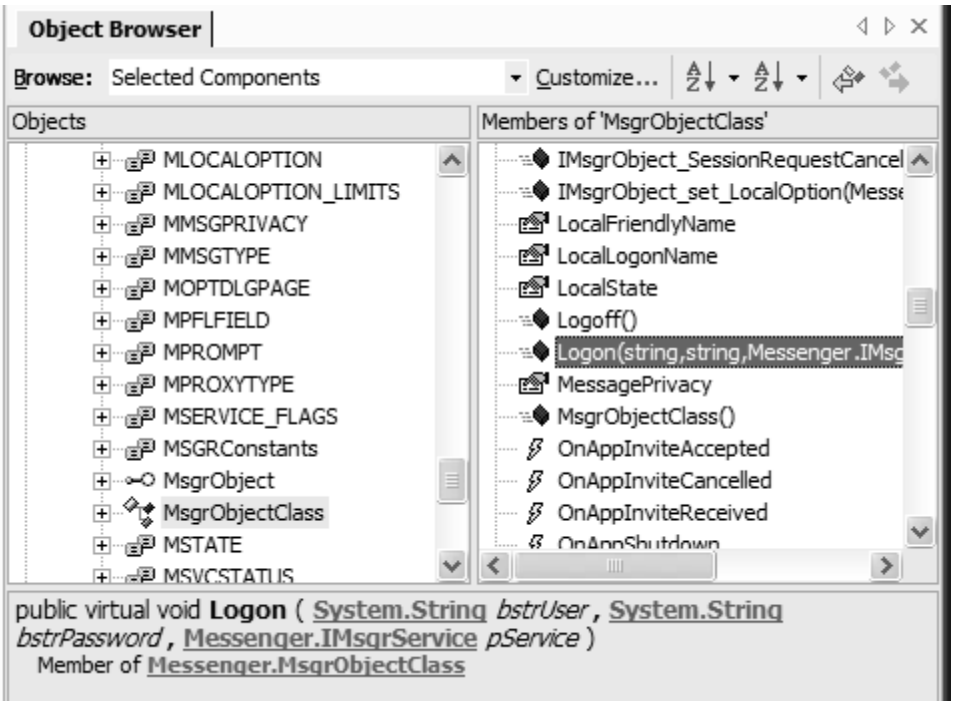


Figure 18-1: The Object Browser is vital to programming COM Interop

So after determining that **MsgrObjectClass** is the type we want to create, it is fairly obvious that our first step is to log on to the IM service and the **MsgrObjectClass.Logon( )** method seems the obvious route. Examining the call in the Object Browser raises the typical COM Interop challenge – the first two arguments **bstrUser** and **bstrPassword** are simple enough, but the third argument is a reference to an **IMsgrService** interface. How do we get that?

Well, the short answer is legwork. The task can be made slightly easier with this utility that uses reflection to determine what properties and methods return the particular **Type** in which you are interested :

```
//:c18:WhatReturns.cs
//Determines what methods return a given type
using System;
using System.Reflection;

public class WhatReturns {
  ///<Summary>
  ///Prints a list of methods in a given assembly
```

```
///that return a given type also in the assembly
///USAGE: WhatReturns MyAssembly MyType
///</Summary>
public static void Main(string[] args){
  if (args.Length != 2) {
    Console.WriteLine(
      "USAGE WhatReturns AssemblyName TypeName");
  }
  new WhatReturns(args[0], args[1]);
}

WhatReturns(string assemblyName, string typeName){
  Assembly asm = Assembly.Load(assemblyName);
  Console.WriteLine("Assembly loaded");
  Type[] types = asm.GetExportedTypes();
  Console.WriteLine(
     "Contains {0} types", types.Length);
  Type targetType = asm.GetType(typeName);
  Console.WriteLine("Target type created");
  Console.WriteLine("Type is: " + targetType);
  foreach(Type type in types){
    TypeReturns(type, targetType);
  }
}

bool TypeReturns(Type type, Type targetType){
  bool atLeastOneThingReturnsType = false;

  PropertyInfo[] props = type.GetProperties();
  foreach(PropertyInfo prop in props){
    bool b = PropertyOfType(prop, targetType);
    if (b) {
      Console.WriteLine(
        " of type " + type.Name);
      atLeastOneThingReturnsType = true;
    }
  }
  MethodInfo[] methods = type.GetMethods();
  foreach(MethodInfo method in methods){
    bool b = MethodReturns(method, targetType);
    if (b) {
```

```
        Console.WriteLine(
          " of type " + type.Name);
        atLeastOneThingReturnsType = true;
      }
    }
    return atLeastOneThingReturnsType;
  }

  bool PropertyOfType(PropertyInfo prop,
    Type targetType){
    Type t = prop.PropertyType;
    bool b = ( t == targetType
      || t.IsSubclassOf(targetType));
    if (b) {
      Console.Write("Property: " + prop.Name);
    }
    return b;
  }

  bool MethodReturns(MethodInfo method,
    Type targetType){
    Type t = method.ReturnType;
    bool b = (t == targetType
      || t.IsSubclassOf(targetType));
    if (b) {
      Console.Write("Returned by: " + method.Name);
    }
    return b;
  }
}///:~
```

With this utility and a bit of sleuthing, we discover that the **MsgrObjectClass** has an **IMsgrServices** property that has a **PrimaryService** property that, in fact, turns out to be the .NET Messenger Service we're interested in. As discussed above, COM methods return **HRESULT**s that are converted into exceptions. This can sometimes cause confusion because **HRESULT** design guidelines are not as strict as exception design guidelines; one should not throw an exception as part of a normal course of execution but as can be seen here **MsgrObjectClass.Logon( )** ends up throwing a **COMException** in the perfectly legitimate scenario in which you're already logged in to the service. The **catch** block in **MessengerSend.Login( )** catches any **COMException**

thrown, but only swallows the particular exception associated with being already logged in. Otherwise, the exception is rethrown.

MessengerSend.FindByFriendlyName( ) attempts to find the **IMsgUser** corresponding to the name passed in as the first argument to the program. Again, getting this to work was a matter of exploratory programming. If an **IMsgUser** is returned by **MessengerSend.FindByFriendlyName( )**, the next step is **MessengerSend.SendMessage( )**. Microsoft Messenger does not appear to provide a stream-based interface to the underlying connection, which is of type **IMsgrIMSession**. Instead, you must use the **IMsgrIMSession.SendText( )** command to send a block of data at a time. The first parameter consists of the header text, which can be **null** (but which turns out *should* be a MIME header – details in the next example).

After compiling this with:

```
csc /reference:Messenger.dll SendMessenger.cs
```

you will have a command-line transmitter for Microsoft Messenger.

Naturally, the next step is to receive messages. Unfortunately, a defect in the current **tlbimp** raises its ugly head. As Figure 18-1 shows, **tlbimp** generates event delegates for the Microsoft Messenger COM Server. Unfortunately, it incorrectly generates **private** helper classes for these events! The end result is that you can write event-handling code and everything compiles fine, but when you run your program, the attempt to set a delegate method in your code generates a **COMException** with **HRESULT** set to 0x80040202.

You can fix this problem with a hack: You disassemble the generated assembly, change the helper class to **public**, and reassemble the assembly. To do this you will need to use the Intermediate Languge assembler and disassembler (**ilasm.exe** and **ildasm**.exe) that come with the .NET Framework SDK.

The first step is to transform the assembly into Intermediate Language (IL) form:

```
ildasm Messenger.dll /out:Messenger.il
```

When you open this (large) file in a programming editor you will see the assembly language of the .NET virtual computer (with "machine-level" support for everything from virtual function calls to **try-finally** blocks). Search for the function:

```
.class private auto ansi sealed
DMsgrObjectEvents_SinkHelper
```

and change the class declaration to **public**:

```
.class public auto ansi sealed DMsgrObjectEvents_SinkHelper
```

Now, save the file, and reassemble it into a binary library:

```
ilasm /dll Messenger.il
```

Obviously, it is to be expected that Microsoft will fix **tlbimp** so that this hack is not needed in the future, but the role of the programmer is to deliver client value as quickly as possible, not to sit around and wait for perfect tools.

With your newly fixed **Messenger.dll** in hand, you can now write event-handlers for Messenger events. In this case, we are simply interested in writing a console program that echoes received text. The corresponding event is **MsgrObjectClass.OnTextReceived( )** and its delegate type is **DMsgrObjectEvents_OnTextReceivedEventHandler**.

The code is straightforward:

```
//:c18:MessengerReceive.cs
//Compile with
//csc /reference:Messenger.dll MessengerReceive.cs
//Echoes MSN Messenger text to console
using System;
using System.Runtime.InteropServices;
using Messenger;
using System.Threading;

class MessengerReceive {
  private static readonly string
  USER="noone@ThinkingIn.Net";
  private static readonly string
  PASS="NotMyPassword";

  [STAThread]
  public static void Main() {
    try {
      MsgrObjectClass msgrObj =
        new MsgrObjectClass();
      msgrObj.OnTextReceived +=
        new
      DMsgrObjectEvents_OnTextReceivedEventHandler(
        TextReceived);
```

```
      try {
        IMsgrServices svcs = msgrObj.Services;
        msgrObj.Logon(
          USER, PASS, svcs.PrimaryService);
      } catch (COMException logonException) {
        if (logonException.Message ==
          "Exception from HRESULT: 0x81000304.") {
          Console.WriteLine("Already logged in");
        } else {
          throw logonException;
        }
      }
      Console.WriteLine("Logged in as " + USER);
      while (true) {
        Thread.Sleep(10000);
      }
    } catch (Exception ex) {
      Console.WriteLine(ex.StackTrace);
    }
  }

  public static void TextReceived(
    IMsgrIMSession session, IMsgrUser user,
    string header, string text, ref bool b) {
    Console.WriteLine(
      "Message from " + user.FriendlyName);
    Console.WriteLine("Header: " + header);
    Console.WriteLine("Contents: " + text);
  }
}///:~
```

After creating a **MsgrObjectClass** object, the **MessageReceiver.TextReceived( )** is set to receive incoming text events. The main thread just idles; when text is received, **MessageReceiver.TextReceived( )** outputs the name of the sender, the header, and the text. A typical run of this might look like:

```
Logged in as lobrien@ThinkingIn.Net
Message from Hotmail
Header: MIME-Version: 1.0
Content-Type: text/x-msmsgsprofile; charset=UTF-8
LoginTime: 1023398920
```

```
EmailEnabled: 1
MemberIdHigh: 72103
MemberIdLow: -1129385323
lang_preference: 1033
preferredEmail: notreallyanaddress@hotmail.com
country: US
PostalCode: 94930
Gender: m
Kid: 0
Age: 38
sid: 507
kv: 2
MSPAuth:
2IZcLlTK1Dj8gTBMWJ*hMRi3e15hDXXJeCyfyRY*GvWn5aV16HcFU!4EMCw
LyTM8telveUq
2uD3PvrE8IwWdbwqg$$


Contents:
Message from Hotmail
Header: MIME-Version: 1.0
Content-Type: text/x-msmsgsinitialemailnotification;
charset=UTF-8


Contents: Inbox-Unread: 113
Folders-Unread: 118
Inbox-URL: /cgi-bin/HoTMaiL
Folders-URL: /cgi-bin/folders
Post-URL: http://www.hotmail.com


Message from bruce
Header: MIME-Version: 1.0
Content-Type: text/x-msmsgscontrol
TypingUser: bruce@ThinkingIn.Net


Contents:

Message from bruce
```

```
Header: MIME-Version: 1.0
Content-Type: text/x-msmsgscontrol
TypingUser: bruce@ThinkingIn.Net


Contents:

Message from bruce
Header: MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8
X-MMS-IM-Format: FN=MS%20Shell%20Dlg; EF=; CO=0; CS=0; PF=0


Contents: The possibilities are endless
```

As you can see, the header information is even more interesting than the message text! After you receive your own profile, you receive information on your Hotmail account, and control messages (so *that's* how the IM client can change its status to "Bruce is typing a message") before ultimately receiving the text itself. One can imagine introducing new MIME content/types to extend the capabilities of the instant messaging realm – sports scores, traffic updates, chess moves. Unfortunately, COM Interop does not work on Microsoft's .NET Compact Framework, so wireless applications will have to rely on other undocumented techniques that are "left as an exercise for the interested student!"

This has been a very interesting discussion.Creating and receiving HTTP requests

Now that we've talked about low-level connections with **TcpClient** and **TcpListener**, and peer-to-peer connections with MSN Messenger, it's time to turn our attention to programming for the World Wide Web. As you know, the World Wide Web is based on the Hypertext Transport Protocol (HTTP), which is an application-level, generic, stateless protocol originally intended for creating distributed hypermedia applications.[5]

Of course, the most common applications that generate HTTP requests are Web browsers, but it is often useful, especially for debugging purposes, to programmatically generate HTTP requests. The .NET Framework SDK makes

[5] Defined at *http://www.ietf.org/rfc/rfc2068.txt*

this easy with the **WebRequest** and **HttpWebRequest** classes in the **System.Net** namespace. The simplest HTTP request just sends an HTTP GET request to a specific Uniform Resource Identifier (URI, the technical term for what are commonly called URLs). An HTTP Server (aka a Web Server) responds with a status code, a header, and then a message body.

This example sends an HTTP GET request for a command-line argument:

```
//:c18:HttpGet.cs
//Echoes result of simple HTTP Get request
//Use: HttpGet http://www.ThinkingIn.Net/index.html
using System;
using System.Net;
using System.IO;

class HttpGet {
  public static void Main(string[] args){
    HttpWebRequest req =
      (HttpWebRequest) WebRequest.Create(args[0]);

    Console.WriteLine("Created, but not connected");
    HttpWebResponse res =
      (HttpWebResponse) req.GetResponse();
    Console.WriteLine("Status: " + res.StatusCode);
    foreach(string key in res.Headers.Keys){
      Console.WriteLine("Header[{0}]:{1}",
        key, res.Headers[key]);
    }

    Console.WriteLine("Contents:");
    StreamReader rdr = new StreamReader(
      res.GetResponseStream());
    Console.WriteLine(rdr.ReadToEnd());
  }
}///:~
```

The way in which you construct an **HttpWebRequest** is similar to the *Builder*-like pattern employed in the **ZALogAnalyzer** example earlier in this chapter: a **string** is passed to a base class, in this case **WebRequest**, which returns an appropriate implementation, in this case an **HttpWebRequest**. The only other type of **WebRequest** in the .NET Framework SDK is a **FileWebRequest** that is created if the argument to **WebRequest.Create( )** starts with "file://," but if

you write a new subtype for, say, "ftp://" URIs, you can call the static **WebRequest.RegisterPrefix( )** method and **WebRequest.Create( )** will thereafter return the new **WebRequest** subtype appropriately.

The request does not actually go out over the network until the call to **WebRequest.GetResponse( )**. Because we know that we're dealing with a request that begins with "http://," we know that we can downcast the **WebResponse** to **HttpWebResponse**. After outputting the **HttpWebResponse.StatusCode**, we write the HTTP headers to the console. The headers are in the **HttpWebResponse.Headers** property and take the form of a **WebHeaderCollection** object that is a subtype of **NameValueCollection**. Finally, the contents of the Web page are echoed to the console.

The flip side of generating an **HttpRequest** is responding to them. The .NET Framework SDK is designed to support Microsoft's Internet Information Server (IIS), and the most direct form of support is the **IHttpHandler** interface, also in the **System.Web** namespace.

IHttpHandler defines just two things: a **bool** property called **IsReusable** that specifies whether the object is capable of servicing another HTTP request. This will depend upon the state of the object that implements **IHttpHandler**. When writing Web programs one component of scalability is minimizing state in your server, so it's generally preferable if your design can result in this property being **true**.

The other component of **IHttpHandler** is the **IHttpHandler.ProcessRequest( )** method, which takes an **HttpContext** object as its lone argument. The most important properties in **HttpContext** are the **HttpRequest** and **HttpResponse** objects (note that these classes are different than the **HttpWebRequest** and **HttpWebResponse** classes that are used at the client).

The **HttpRequest** contains a large number of properties, the most commonly used of which is the **HttpRequest.Form** property, which returns a **NameValueCollection** whose keys are the form variable names and whose values are the form values as **string**s.

This example responds with the mime types that are accepted by the client, as well as any form variables:

```
//:c18:SimpleHandler.cs
//A simple form handler for integration with IIS
```

```
//Compile with:
//csc /target:library SimpleHandler.cs
using System;
using System.Web;

class SimpleHandler : IHttpHandler {
  public bool IsReusable{
    get { return true;}
  }

  public void ProcessRequest(HttpContext context){
    HttpResponse res = context.Response;
    res.Write(
      "<html><body><h1>Mime types</h1><ul>");
    HttpRequest req = context.Request;
    foreach(string mimetype in req.AcceptTypes){
      res.Write("<li>" + mimetype + "</li>");
    }
    res.Write("</ul>");
    if (req.Form.Count > 0) {
      res.Write("<h1>Form variables</h1><ul>");
      foreach(string varName in req.Form){
        res.Write("<li>" + varName + ":" +
          req.Form[varName] + "</li>");
      }
      res.Write("</ul>");
    }
    res.Write("</body></html>");
  }
}///:~
```

After getting the **HttpResponse** object from the **HttpContext** argument, **SimpleHandler.ProcessRequest( )** begins the task of writing HTML. Writing the HTTP response header is not necessary, but if you need to customize the header, you can do so by setting a variety of properties of the **HttpResponse** and calling methods such as **HttpResponse.SetCookie( )**.

As part of the HTTP request message, the client sends a list of MIME types that it can accept. This is typical of the sort of property that the **HttpRequest** exposes but which most of the time you do not care about. In this case, the types are written to the output as an HTML unordered list. The method then checks if there are any form variables and, if so, outputs them, also as an unordered list.

Installing the program for IIS takes a few additional steps. First, compile the program to a library assembly. To better illustrate IIS' configuration, specify the name of the assembly with:

```
csc /target:library /out:MyHandlers.dll SimpleHandler.cs
```

Then, decide where on the Web site and by what name you wish to expose the functionality. By convention, the name of a .NET program exposed on a Web site should be ***Xxx*.aspx**. So let's say that we wish to expose the **SimpleHandler** type in the **MyHandlers.dll** assembly on the local machine's IIS so that it could be reached by:

```
http://localhost/TargetName.aspx
```

To let IIS know that when it receives such a request it should load and call our program, you have to edit the **Web.config** file in the Web directories (which are often installed at **c:\inetpub\wwwroot**). The **Web.config** file is an XML document with many options, adding **IHttpHandler**s is done by manipulating the **<httpHandlers>** element that is a child of **<system.web>**:

```
<configuration>
  <system.web>
    <!-- etc -->
    <httpHandlers>
      <add path="TargetName.aspx" verb="*"
           type="SimpleHandler,MyHandlers"/>
    </httpHandlers>
    <!-- etc -->
  </system.web>
</configuration>
```

The **<httpHandlers>** element does not seem to be part of the default **Web.config**, so you will probably have to add it. The actual configuration of the **SimpleHandler** object is done with the **<add>** element.

The **path** attribute specifies what URLs should be handled by this **IHttpHandler**. The **path** attribute accepts wildcards so, for instance, **path="*.mytype"** could be used to handle all requests for a particular type. The **HttpRequest.Url** (that's an 'l' at the end, not an 'i') could then be used to retrieve an object of type **Uri** ('i' not 'l') to determine exactly what "mytype" to retrieve.

The **verb** attribute specifies which HTTP request types (GET, POST, etc.) this type should handle, in this example we are handling all of them. Finally, the **type** attribute specifies that our **IHttpHandler** is of type **SimpleHandler** as

defined in the **MyHandlers** assembly (notice that one does *not* specify the ".dll" extension).

After saving this file, IIS will attempt to handle the desired URI with our type. IIS loads the assembly from the \bin subdirectory of the directory in which the **Web.config** file has been saved (if the above was saved as c:\inetpub\wwwroot\Web.config, IIS would attempt to load the **SimpleHandler** type from c:\inetpub\wwwroot\bin\MyHandlers.dll).

When accessed from this Web page:

```
<html>
<body>
<form action="http://localhost/TargetName.aspx"
method="post">
<input type="text" name="textfield" size="35">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

The output will look like Figure 18-2

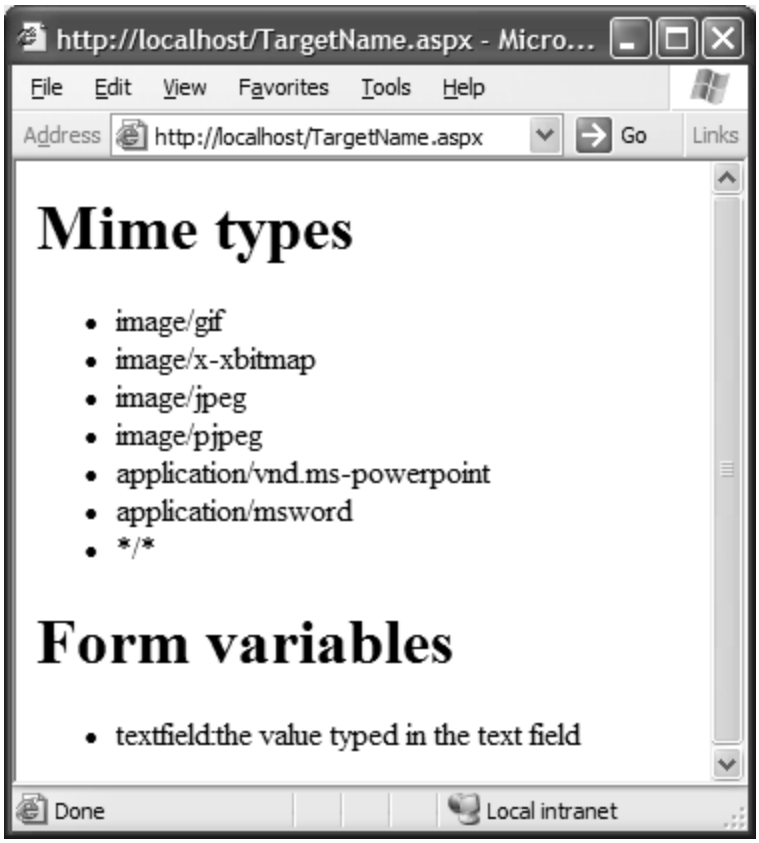


Figure 18-2: The SampleHandler in action

Asynchronous Web requests

A previously undiscussed aspect of the .NET Frameworks IO streams is support for *asynchronous IO*, in which the calls to read from or write to a stream do not block. Asynchronous IO uses an asynchronous design that Microsoft is advocating for widespread use: there is a begin operation and an end operation and these activate asynchronous delegates of type:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

The **IAsyncResult ar** has fields that contain:

- An **Object AsyncState** that is the domain object that should reflect any state that is needed for the callback to function
- A **WaitHandle AsyncWaitHandle** that can be used to synchronize multiple threads involved in the asynchronous operation

- **CompletedSynchronously** and **IsCompleted**, both **bool**s, that specify whether the begin operation completed synchronously and whether the end operation has completed at all**.**

In the case of a **WebRequest**, the synchronous call to **WebRequest.GetResponse( )** has the corresponding asynchronous begin and end methods of **WebRequest.BeginGetResponse ( )** and **WebRequest.EndGetResponse( )**. To create a truly asynchronous Web request, you need to use these methods as bookends for an asynchronous read of the response's **Stream** using **Stream.BeginRead( )** and **Stream.EndRead( )**.

To download a large file asynchronously, you could, of course, use **Stream.Read*Xxx*( )** methods in your own **ThreadStart** delegate. But the asynchronous IO library in .NET is likely to be more efficient and robust. This example shows a class that asynchronously downloads a given URI and passes the results back using a custom event.

```
//:c18:AsyncWeb.cs
//Demonstrates asynchronous network IO
using System;
using System.Text;
using System.Net;
using System.IO;
using System.Threading;

public delegate void PageDownloadedHandler(
  object sender, PageDownloadedEventArgs mea);

public class PageDownloadedEventArgs : EventArgs {
  private string msg;
  public string Message{
    get { return msg;}
    set { msg = value;}
  }
  public PageDownloadedEventArgs(string msg){
    this.msg = msg;
  }
}

class AsyncDownloader {
  WebRequest req;
```

```
WebResponse resp;
StringBuilder responseBuilder =
  new StringBuilder();
Stream responseStream;
Decoder streamDecode = Encoding.UTF8.GetDecoder();
static readonly int BUFFER_SIZE = 256;
byte[] buffer = new byte[BUFFER_SIZE];

public event PageDownloadedHandler PageDownloaded;

protected void OnPageDownloaded(
  PageDownloadedEventArgs ea){
  Console.WriteLine("Page downloaded");
  if (PageDownloaded != null) {
    PageDownloaded(this, ea);
  }
}

public AsyncDownloader(string uri){
  Console.WriteLine("AsyncDownloader({0}", uri);
  req = WebRequest.Create(uri);
  Console.WriteLine("AsyncDownloader() ends");
}

public void Run(){
  Console.WriteLine("Run()");
  //Set request callback
  AsyncCallback beginReqCallback =
    new AsyncCallback(BeginRequestCallback);
  req.BeginGetResponse(beginReqCallback, null);
  Console.WriteLine("Run() ends");
}

//BeginGetResponse delegate
private void BeginRequestCallback(IAsyncResult ar){
  Console.WriteLine("BeginRequestCallback()");
  resp =
    (HttpWebResponse) req.EndGetResponse(ar);
  responseStream = resp.GetResponseStream();
  AsyncCallback strRdCb =
    new AsyncCallback(StreamReadCallback);
```

```
    responseStream.BeginRead(
      buffer, 0, BUFFER_SIZE, strRdCb, null);
    Console.WriteLine("BeginRequestCallback ends");
  }

  private void StreamReadCallback(
    IAsyncResult asyncResult){
    Console.WriteLine("StreamReadCallback()");
    int read = responseStream.EndRead(asyncResult);
    if (read > 0) {
      Console.WriteLine("Read {0} bytes", read);
      Char[] charBuffer = new Char[BUFFER_SIZE];
      int len = streamDecode.GetChars(
        buffer, 0, BUFFER_SIZE, charBuffer, 0);
      String str = new String(charBuffer, 0, len);
      //Console.WriteLine("Read: " + str);
      lock(this){
        responseBuilder.Append(str);
      }
      AsyncCallback moreToGoCb =
        new AsyncCallback(StreamReadCallback);
      responseStream.BeginRead(
        buffer, 0, BUFFER_SIZE, moreToGoCb, null);
    } else {
      Console.WriteLine("No more to read");
      responseStream.Close();
      PageDownloadedEventArgs ea =
        new PageDownloadedEventArgs(
          responseBuilder.ToString());
      OnPageDownloaded(ea);
    }
    Console.WriteLine("StreamReadCallback() ends");
  }
}

class AsyncClient {
  static Thread mainThread;

  public static void Main(){
    AsyncDownloader ad = new AsyncDownloader(
      "http://www.ThinkingIn.Net");
```

```
    ad.PageDownloaded +=
      new PageDownloadedHandler(
        MyDownloadEventHandler);
    Console.WriteLine("Before async call happens");
    ad.Run();
    Console.WriteLine("After async call");
    mainThread = Thread.CurrentThread;
    try {
      Thread.Sleep(30000);
    } catch (ThreadInterruptedException ) {
      Console.WriteLine("Finished before timeout");
    }
  }

  static void MyDownloadEventHandler(
    object src, PageDownloadedEventArgs ea){
    Console.WriteLine("Got a page of {0} chars",
      ea.Message.Length);
    mainThread.Interrupt();
  }
}///:~
```

The first things we define are the delegate and event arguments for the "page downloaded" event, **PageDownloadedHandler** and **PageDownloadedEventArgs**. The **PageDownloadedEventArgs** will contain the downloaded page in its **Message** property.

We then begin defining the **AsyncDownloader** class. All three important elements of an HTTP request are stored in instance variables: the **WebRequest req**, the **WebResponse resp**, and the **Stream responseStream**. Then, we define a **byte[] buffer** to hold the raw response data and to convert the raw response into a single **string** we declare and initialize **StringBuilder responseBuilder** and **Decoder streamDecode**.

The **event PageDownloaded** is a multicast delegate that can be subscribed to by anyone interested in being notified when the asynchronous page download is complete. Following the standard .NET idiom for writing events (see Chapter 14), we place the raising of the **PageDownloaded** event inside a **protected** method called **OnPageDownloaded** that simply determines if anyone has subscribed to the **PageDownloaded** event and, if so, activates the delegate, passing in **this** as the source of the event and the passed-in event arguments.

The **AsyncDownloader( )** constructor takes a **string uri** as a parameter and uses that to initialize the instance variable **WebRequest req.** The next method **Run( )** actually begins the process of an asynchronous download. First, we specify that **BeginRequestCallback** should be used as our first **AsyncCallback** delegate. **WebRequest.BeginGetResponse( )** takes an **AsyncCallback** and any **object** that encapsulates important state that the callback may need to determine its context. Because our **BeginRequestCallback** delegate is a method of **this**, when the callback occurs we will have all the context we need and so, rather than pass in something meaningless, explicitly pass **null**. If this was not **null**, the **object** we passed in would be available to the **AsyncCallback** in the **IAsyncResult** argument's **AsyncState** property.

When you run this program, the call to **BeginGetResponse( )** does not block, so "Run() ends" is printed immediately after "Run()." But at some point, the Web request begins and the **BeginRequestCallback** gets called. **WebRequest.EndGetResponse( )** is used to end the asynchronous production of the **WebResponse resp**, but that's only half our asynchronous challenge, because now we must asynchronously read the response stream (well, we *could* read it synchronously, but that would defeat the whole purpose of the exercise, as it's the data of the response body that's large, not the data making up the response header).

In order to read the stream asynchronously, we need another **AsyncCallback** delegate, this time using the method **StreamReadCallback**. We call **Stream.BeginRead( )** with a reference to **this.buffer,** the starting position in **buffer** and size to write (0 and **BUFFER_SIZE** respectively), and the **AsyncCallback** to **StreamReadCallback**. Again, we're calling back to **this**, so we pass **null** for the **IAsyncResult**'s **AsyncState.**

StreamReadCallback( ) gets called when the **responseStream** has filled the buffer or finished reading. If data was read, the bytes in the **buffer** are converted into a **Char[ ]** array by the **Decoder streamDecode,** then into a **string**, and then the **string** is appended to the **StringBuilder responseBuilder** (this data manipulation is placed in a **lock** clause to ensure the thread integrity of the **buffer** and **responseBuilder** objects). Since more data might remain, a new **AsyncCallback** is instantiated on **StreamReadCallback**. Remember that **Stream.BeginRead( )** doesn't block and returns immediately, so even though this may look like a recursive call, it's not.

Jumping back towards the top of **StreamReadCallback( ),** when the **Stream** has no more data, the call to **Stream.EndRead( )** will return 0 bytes read and

control will move to the **else** part of the clause. The final step in the asynchronous read of the page is to close the **responseStream,** create a new **PageDownloadedEventArgs** with all the data that's accumulated in the **StringBuilder responseBuilder**, and call **AsyncDownloader.OnPageDownloaded( )**.

The **AsyncClient** class creates an **AsyncDownloader** class to read *www.ThinkingIn.net*. The **PageDownloaded** event is subscribed to by the static method **MyDownloadEventHandler.** After calling **AsyncDownloader.Run( )**, the main thread is sent to sleep for 30 seconds (in a graphical application, one can imagine a splash screen, progress bar, or hourglass appearing). When **MyDownloadEventHandler** is called by the **AsyncDownloader**, it calls **Thread.Interrupt( )** on the main thread, so even if the 30 seconds has not elapsed, the main thread is interrupted when the read completes.

When you run this program, the most striking thing is that **AsyncDownloader.StreamReadCallback( )** many times before it returns even once, indicating that it is called asynchronously by the thread that is monitoring the response stream. As mentioned previously, the asynchronous IO design of the library is almost undoubtedly more efficient than what one would casually implement.

From Web programming to Web Services

Web Services are *headless* applications whose input and output are sent with standard Internet protocols. By "headless," we mean that Web Services do not have user interfaces (of course, a user interface is ultimately needed, but it's created by something other than the Web Service). Instead of objects, or HTML (which is a user-interface specification), Web Services use XML documents for input and output. How the XML input is generated and how the XML output is consumed are not the concerns of the Web Service.

By "standard Internet protocols," we give ourselves a *lot* of wiggle room. As "Web Services" becomes the buzzword du jour, various vendors compete by looking at their own feature list and saying either "If it doesn't support this protocol, it's not complete!" or "That protocol is a frivolous extension designed to lock you in to their solution!"

More significantly, there are competing standards for several core functions, including the means by which a procedure is called and the means by which Web

Services are described and discovered. The former is a critical decision for all Web Service development, but .NET is firmly in the camp of Simple Object Access Protocol (SOAP). The latter will *eventually* become critical to the development of new channels but today virtually all Web Service development is done in direct partnership with the service's consumers and description and discovery is done, not automatically, but via meetings, emails, and project Wikis.[6]

Insanely simple Web services

The "Hello, C#" programs that started this book included one associated with the **WebMethodAttribute**. As part of its compilation process, Visual Studio .NET recognizes this attribute and automatically exposes on IIS via SOAP. It turns out, though, that exposing methods with **[WebMethod]** can be even easier.

IIS handles URIs ending in **.asmx** with Web Service tools. The file must have a special Web Service declaration line at the beginning, but other than that, exposure as a Web Service is strictly a matter of attributes. This example returns the time on the current server:

```
//:c18:WhatsTheTime.asmx
//Web Service example
//Save in /inetpub/wwwroot as WhatsTheTime.asmx
//Access via http://localhost/WhatsTheTime.asmx
<%@ WebService Language="C#" Class="WhatsTheTime" %>

using System;
using System.Web.Services;

[WebService(Namespace="http://www.ThinkingIn.Net/")]
class WhatsTheTime{
  [WebMethod] public DateTime Time(){
    return DateTime.Now;
  }
}///:~
```

The file *must* be saved as ***Xxx*.asmx,** it *must* include the <%@ ... %> declaration, exposed methods *must* be both **public** and associated with

[6] A Wiki is an openly editable Web site and is a must for any development team's intranet. The original "WikiWikiWeb" was created by Ward Cunningham to house a repository of design patterns for programming. See *http://c2.com/cgi-bin/wiki?WikiWikiWeb*

WebMethodAttribute, and you *must* serve it from IIS running on a system with the .NET Framework installed.

IIS takes care of generating a service description in Web Services Description Language (WSDL) and generating a sample page that shows you how the Web Service can be called with HTTP POST and GET requests as well as with SOAP. The auto-generated page also allows you to test the Web Service interactively as shown in Figure 11-3.

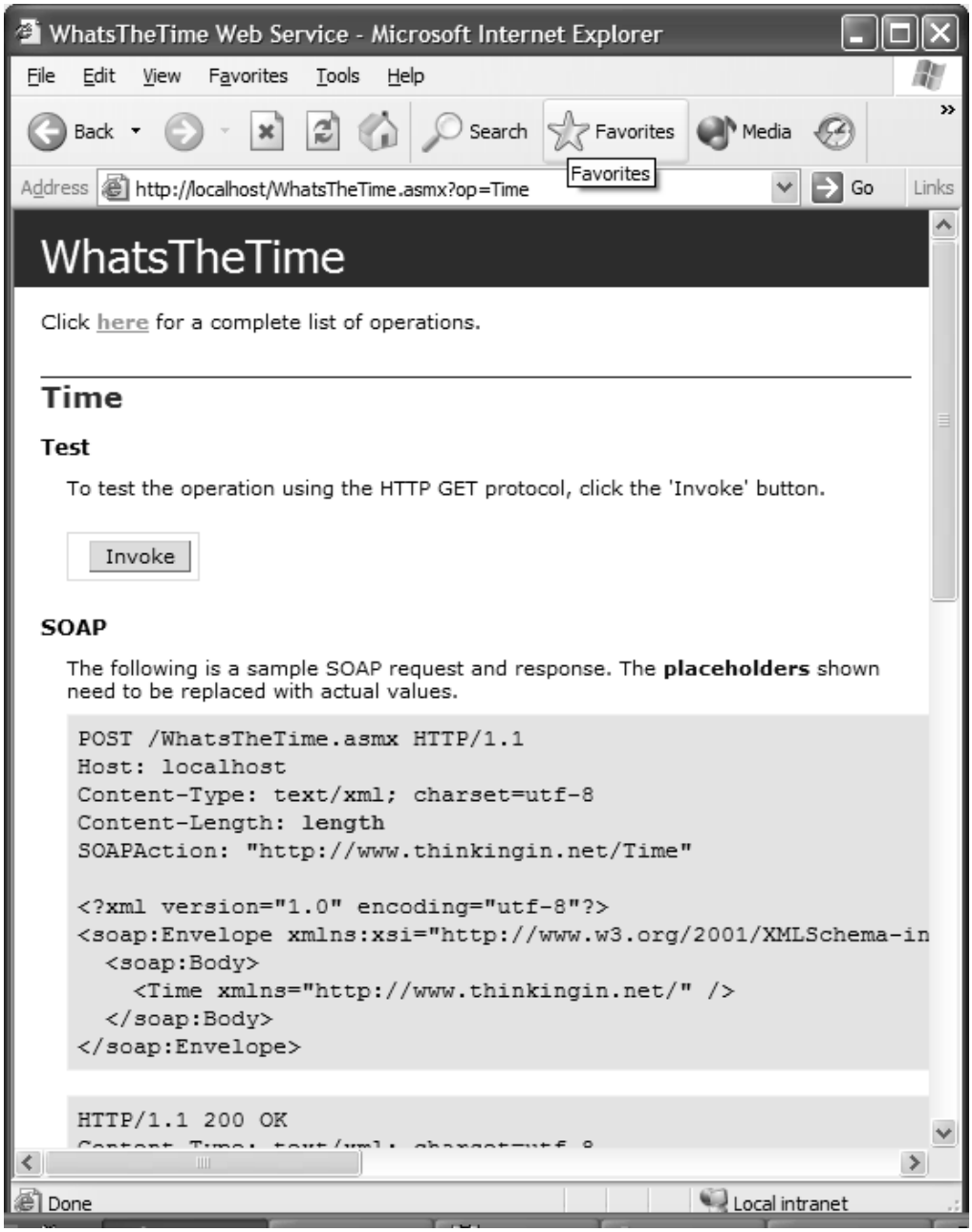


Figure 18-3: This page was created automatically

As tremendous as this is, most of the time you will not want to put all of your code into a single **.asmx** file but will prefer to load them from an assembly.

In that case, you still need an **.asmx** file, but the **WebService** declaration takes a slightly different form:

```
<%@ WebService Language="C#" Class="Type,Assembly" %>
```

This time, the **WebMethodAttribute** methods in the **Type** type will be loaded from the **Assembly** assembly in the \bin subdirectory. When you use Visual Studio .NET to create a Web Service project, this is the type of **WebService** declaration that it generates.

The <%@ ... @> block is an ASP.NET declaration. ASP.NET is, like ADO.NET and XML, a technology that intersects with C#, but is not a subset. ASP.NET is a compiled environment, programmed by embedding blocks (such as the WebService declaration above) in HTML code. ASP.NET is beyond the scope of this book, but we'll mention one more ASP.NET facility that is relevant to many Web Service development scenarios, which is state management in Web Services

Maintaining state

As the **WhatsTheTime** example demonstrated, any method can be exposed as a Web Service. However, there is a **WebService** class in the **System.Web.Services** namespace that provides additional functionality that is often useful. In particular, **WebService** provides a **Application** property of type **HttpApplicationState** and a **Session** property of type **HttpSessionState**. Both of these properties are derived from **NameObjectCollectionBase** (see chapter ir10). There purpose is to maintain conversational state across, respectively, the application and session lifecycles. The application lifecycle begins when the **WebService** class is first loaded and persists until the Web server shuts down; it is analogous to static object data. The session lifecycle begins when a particular client hits the server and, by default, works via cookies. If cookies are not available, session state can still be stored via URL rewriting. The cookie is not used to store state, but is used as a key to server-side storage. **HttpSessionState** is analogous to object data.

Web services vs. Web APIs

One of the authors (Larry) has spent most of the past five years actually developing various XML-over-HTTP services for a variety of clients. One characteristic of the successful development projects is that they did not expose an Application Programming Interface, but rather operated in a stateless manner to expose complete, use-case-based value to the client program. An API is really a programmer's viewpoint of a system and often requires sequential association (see page 335); in other words, method **Y( )** is usually a fine-grained atom of functionality that may have as a precondition that method **X( )** has already been

called. You can write such a system and expose it over the Web and call it a Web Service, but just as we've seen that the benefits of object orientation do not come from the simple availability of classes and objects, so too the benefits of Web Services do not come from the simple availability of XML remote procedure calls.

The benefits of Web Services really kick in when the exposed functions individually embody user-meaningful chunks of data. These types of Web Services are sometimes described as *document-centric* because the idea is that the XML queries and responses are complete and self-contained documents.[7]

For instance, when you reserve an airline ticket with a travel agent using a reservation terminal, this is how the sequence goes:

1. "Show me the flights from NYC to SFO on August 1."
2. "Give me two unrestricted seats in the economy class of the flight shown on the third line of the availability response."
3. "Show me the flights from SFO to NYC on August 8."
4. "Add two unrestricted economy-class seats for the flight shown on the second line of the response."
5. "Give me the lowest fare compatible with the dates and the desired travel restrictions, converting the unrestricted reservations into restricted ones."
6. "Here's the traveler information…"
7. "Ticket it using payment method ABC."

Now, you *could* expose each of these steps as individual function calls with sequential association (adding seats and traveler information to a growing itinerary). However, if you were programming for the document-centric Open Travel Alliance XML specification, you would say something more like:

1. "What is availability for NYC-SFO on August 1?"
2. "What is availability for SFO-NYC on August 8?"

---

[7] Yes, we have to admit that what we're advocating is a usage-centered object-oriented document-centric outward-looking service built with a risk-driven quality-centric use-case-oriented team-based approach.

3. "What are the available pricing codes for two tickets in economy class for the itinerary SomeAir flight 12 on August 1, SomeAir flight 14 on August 8 (NYC-SFO-NYC)?"

4. "Ticket these two travelers with pricing code XYZ and payment method ABC on SomeAir 12 on August 1 and SomeAir14 on Aug 8."

This document-centric approach is better for several reasons. Both approaches ultimately have the same number of alternate scenarios ("What if the user doesn't like any of the shown flights? What if they want to split the price between a credit card and frequent flyer miles? What if they are flexible in terms of travel dates?"), but in the document-centric approach, the effect of these alternate scenarios is confined to individual steps. Thus, if you were generating incorrect itinerary prices, in the API-style Web Service you would have a difficult time saying with confidence that things were *x*% complete or know exactly how to allocate effort. In the document-style approach, you could say with confidence "We know availability works and we know that if we ticket with pricing code XYZ, we get the correct itinerary price, so the problem must be in the third step, which was Dumb Dan's responsibility. We're going to have Arnold the Amazing team up with Dumb Dan to see if we can't shake that out and put Productive Paula on the 'flexible flight dates' use-case.'"

Note that the document-centric approach allows for:

- Better managerial control in terms of both task completion and resource allocation
- Better client communication
- Better fault isolation
- Better change-request impact analysis
- Isolation of business logic concerns from presentation concerns

And also notice that none of these benefits are guaranteed just because the project is using XML or SOAP or WSDL or any other technologically-oriented thing. Just as C# facilitates but does not guarantee high-productivity programming, and just as object-orientation facilitates highly-cohesive, loosely-coupled design, so too the document-centric approach to Web Services facilitates, but does not guarantee, some project characteristics that would be beneficial in *any* situation.

Even in the document-centric approach, there may of course be temporal requirements – a travel Web site may very well assume that all users will do an availability request before issuing a ticketing request. But in the document-

centric view of Web Services, such things are responsibilities of the presentation layer, which include the ASP.NET portion of the system. *Not all server-side programming is "Web Service" programming*. In a Web Services project you will often conceptually have three different servers as shown in Figure 18-4: one for serving static and dynamic Web content, one for controlling the database, and a third for receiving and responding to XML-based Web Service requests.

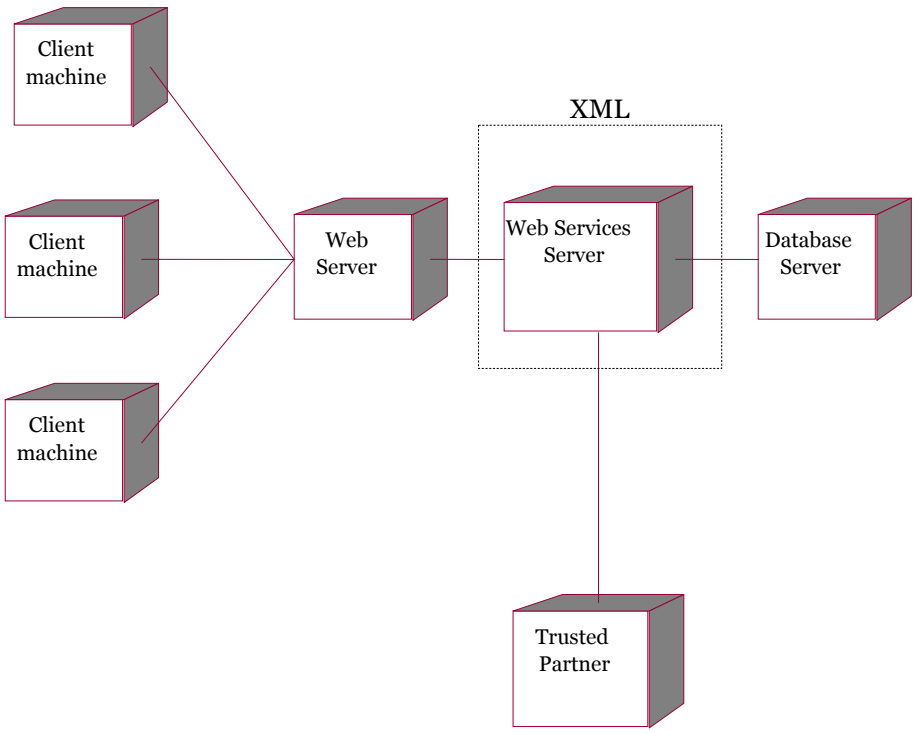


Figure 18-4: Not all server-side programming is Web Service programming

Whether these three servers are on three different boxes is strictly an IT issue relating to performance. Also, although "XML in, XML out" is the conceptual mantra of Web Services, we'd recommend using an ADO.NET native **IDataAdapter** if possible; .NET's abstraction of relational, XML, and object data (as discussed in Chapters 10 and 17) provides a thorough safety-net if a pure XML view of the data is required.

Consuming Web services

One of the impressive capabilities of the **WebMethodAttribute** class is that it can generate a formal description of the method in an XML format known as *Web Services Description Language* (WSDL). While WSDL does not yet have the broad support that SOAP does, the sheer market power of .NET makes its success somewhat of a fait accompli. That is because the .NET Framework SDK ships with a tool, **wsdl.exe**, that takes a WSDL description and generates source code for a class that can serve as a *proxy* for the Web Service. The proxy class can invoke the Web Service, interpret the results, and return them as native objects. In other words, the **WebMethodAttribute** and the **wsdl.exe** tool make the creation of Web Services truly trivial, as the **WebMethodAttribute** both exposes the service on the server and generates the WSDL that the **wsdl.exe** tool can use to generate the proxy class on the client.

If you run:

```
wsdl http://localhost/WhatsTheTime.asmx?wsdl
```

it will generate the following as **WhatsTheTime.cs**:

```
//------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.3705.209
//
//     Changes to this file may cause incorrect
//     behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
// Reformatted for book display
//------------------------------------------------

//
// This source code was auto-generated by wsdl,
// Version=1.0.3705.209.
//
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;
```

```
using System.Web.Services.Description;

/// <remarks/>
[DebuggerStepThroughAttribute()]
[DesignerCategoryAttribute("code")]
[WebServiceBindingAttribute
   (Name="WhatsTheTimeSoap",
    Namespace="http://www.ThinkingIn.Net/")]
public class WhatsTheTime : SoapHttpClientProtocol {

    public WhatsTheTime() {
        this.Url =
          "http://localhost/WhatsTheTime.asmx";
    }
    [SoapDocumentMethodAttribute
      ("http://www.ThinkingIn.Net/Time",
RequestNamespace="http://www.ThinkingIn.Net/",
ResponseNamespace="http://www.ThinkingIn.Net/",
Use= SoapBindingUse.Literal,
ParameterStyle= SoapParameterStyle.Wrapped)]
    public System.DateTime Time() {
        object[] results =
          this.Invoke("Time", new object[0]);
        return ((System.DateTime)(results[0]));
    }

    /// <remarks/>
    public System.IAsyncResult BeginTime(
      System.AsyncCallback callback,
      object asyncState) {
        return this.BeginInvoke(
          "Time",
          new object[0], callback, asyncState);
    }

    /// <remarks/>
    public System.DateTime EndTime(
      System.IAsyncResult asyncResult) {
        object[] results =
          this.EndInvoke(asyncResult);
        return ((System.DateTime)(results[0]));
```

```
    }
}
```

As you can see, the generated **WhatsTheTime** class descends from **SoapHttpClientProtocol** from the **System.Web.Services.Protocols** namespace and relies on that class's methods to call the Web Service either synchronously with **SoapHttpClientProtocol.Invoke( )** or asynchronously with **SoapHttpClientProtocol.BeginInvoke( )** and **EndInvoke( )**. You can see that the **WhatsTheTime.Time( )** method has precisely the same signature (no arguments and returns a **DateTime**) as the original method in the **WhatsTheTime.asmx** Web Service. Thus, using this class is as simple as:

```
//:c18:WebServiceClient.cs
//Compile with proxy class
//csc WhatsTheTime.cs WebServiceClient.cs
//Or, if proxy has been compiled into a library
//csc /reference:WhatsTheTime.dll WebServiceClient.cs
using System;

class WebServiceClient {
  public static void Main(){
    WhatsTheTime clientProxy = new WhatsTheTime();
    DateTime now = clientProxy.Time();
    Console.WriteLine(now.ToString());
  }
}///:~
```

The call **clientProxy.Time( )** is actually a remote procedure call to the **WhatsTheTime** Web Service, but as you can see, the code makes the process totally transparent.

Modifying XML returns

By associating **WebMethodAttribute** with a method, you aren't achieving the most common goal for Web Services, which is adherence to a vertical industry XML specification. As discussed in Chapter 17, the **[XmlElement]** attribute can be used to control the XML data returned. This example returns XML in RDF Site Summary (RSS) format, the popular language of blogs?.

```
<!-- :c18:Rss.asmx -->
<%@ WebService Language="C#" Class="rss" %>

using System;
using System.Web.Services;
```

```
using System.Xml.Serialization;

[WebService(Namespace="http://www.ThinkingIn.Net/")]
public class rss{
  [XmlAttribute("version")]
  public double version = 0.92;

  public Channel channel = new Channel();
  public Item item = new Item();

  [WebMethod]
  public rss GetBlog(){
    return new rss();
  }
}

public class Channel{
  public string title="My blog";
  public string link = "http://www.ThinkingIn.Net/";

  [XmlElement(ElementName="description")]
  public string desc =
    "By using [XMLElement] and [XmlAttribute], you"
    + " can generate standard-compliant XML easily";
}

public class Item{
  [XmlElement(ElementName="description")]
  public string desc = "An item in the blog";
}///:~
```

The example is another "inline" Web Service class, as opposed to being placed in a library and primarily consists of a number of **public** variables that would normally have dynamic content and be encapsulated as properties. By default, public data is serialized as child elements with the name of the exposed property or field, but the example fine-tunes the result to adhere to RSS. The **double version** field is marked with **[XmlAttribute]** and the two **desc** fields are output as "description" elements. One cannot apply the **XmlElementAttribute** to a class name, though, and the root element of an RSS document is **<rss>** in lowercase, so we were forced to break the C# naming convention and call our **class rss** as well.

The result of this Web Service is a valid RSS document:

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="0.92" xmlns="http://www.ThinkingIn.Net/">
  <channel>
    <link>http://www.ThinkingIn.Net/</link>
    <description>By using [XMLElement] and [XmlAttribute],
you can generate standard-compliant XML
easily</description>
  </channel>
  <item>
    <description>An item in the blog</description>
  </item>
</rss>
```

Summary

In C# and .NET, network and Web programming is just a variation on themes developed throughout this book: object-orientation, separation of concerns, and increasing levels of abstraction. ?The Internet has transformed the computer and the subject of software development from primarily calculation-based to primarily communication-based. The .NET Framework allows you to communicate in any number of ways: with full access to low-level socket programming, using higher-level network libraries, or on top of transport protocols such as HTTP or even MSN Messenger or other Instant Messaging protocols. XML, as discussed in the previous chapter, provides a convenient standard way to package data for interchange between programs. SOAP, an XML specification for invoking behavior, and WSDL, which describes the data format provided by a Web-facing service, makes it even easier for programs to interact.

Web Services are not a silver bullet. It is possible to write a terrible API and expose it over the Web. It is possible to write an unstable system and attempt to deploy it on your server. It is possible to write a hairball of an application that is impossible to modify over time. But it's also possible to avoid these things. Web Services have *a tendency* to promote coarse-grained, usage-centered scenarios that provide "atoms of user functionality." They have *a tendency* to promote a tiered separation of concerns: Web Service clients that deal with the computer-human interface, Web Service tiers that concern themselves with business rules, and persistence tiers that concern themselves with data storage. These two tendencies, coarse-grained documents and

tier separation, *may be* exploited by savvy software project managers to control risk, bring services to market faster, and enhance quality.

This summary has been almost completely buzzword-compliant; the issues relating to Web Services are not really at the language-level, but in the softer realms of product and project management. This book starts with the premise that the language you think in channels your thoughts and that C# is language that facilitates the rapid development of robust and scalable programs that deliver client value and can be modified easily. If you have worked your way to here, we hope that those words have become more meaningful to you, that they now carry connotations of objects and architectures, practices and patterns. There is, of course, much more to professional software development than competence in a language, design methodology, or class library, but we said early on that the concerns of this book stop at the edge of the screen.

We hope that we have helped you develop some amount of confidence in your ability to tackle a problem with C# and the .NET Framework. We hope that thinking in and expressing a solution in C# has started to become second nature to you. The more you work in C#, or any language, the more you will internalize it, and the more you will be able to say precisely what you mean, the more you will be able to express subtleties and, in the expression, discover and define your unique voice. The programming world is filled with brilliant people, infinite potential, and endless mysteries.

Those who can, code.

Exercises

1. Using one the symmetric cryptographic resources of the .NET IO library (as described in chapter 12), modify the **JabberServer** and **JabberClient** programs so that the data they transmit is encrypted using a pre-determined key.

2. Modify exercise 3 so that **JabberClient** and **JabberServer** do not need a pre-determined key. To do this, the **JabberServer** should respond to an initial connection by sending a public key. **JabberClient** generates a random session key that will be used for subsequent communication. **JabberClient** encrypts this key with **JabberServer**'s public key and sends the result to **JabberServer**. **JabberServer** decrypts this message to find the session key. Further communication between **JabberClient** and **JabberServer** should use symmetric cryptography with the session key as the encoding/decoding key.

3. Study the Builder design pattern and compare it to the technique used in this chapter in the **whois** example. Identify circumstances where "classic" Builder would be superior to this method and vice versa.

4. Write your own graphical interface to the MSN Messenger interface.

5. Using the Markov generator from exercise 131, write an MSN chatbot.

6. Write an **IHttpHandler** that stores submitted email addresses in a database.

7. Write an **IHttpHandler** that returns the accumulated email addresses from the previous exercise as an XML document to be displayed in a Web browser.

8. Add an XSLT processing instruction to transform the XML returned from the previous example into XHTML.

9. (Advanced) Using everything you have learned in this book and investigating the **System.Web.Mail** namespace, write a .NET Web Service that provides a mailing list manager.

10. (Advanced) Write a client to your .NET Web Service using a non-.NET language such as Java. Demonstrate interoperability.

11. (Advanced) Write a Web Service that uses the MSN Messenger network to create ad hoc computing grids. Investigate and implement a computationally expensive task (fractal generation, large-number factoring, global climate modeling, etc.) with a custom user interface.

12. (Advanced) Write the coolest thing ever.

A: C# for Visual Basic Programmers

As a group, Visual Basic programmers have the most to gain by moving to C#. While it's always treacherous to make broad pronouncements about a population of several million people, it should be safe to say that Visual Basic programmers have chosen to buy into Microsoft's platform strategies. It may also be safe to say that the majority of Visual Basic programmers have consciously traded some amount of generality in their career for the immediate productivity gains they perceive in Visual Basic. Visual Basic has always been the easiest way to develop Windows programs and, over the years, has consistently made it easy to access operating system features. C# is perhaps not *quite* as easy as Visual Basic, but the advantages of the C-derived syntax, full and consistent object orientation, and the type system more than make up for the incremental difficulty.

The first thing that will strike a VB programmer using Visual Studio .NET is – nothing. In the opening moments of programming a Windows application, C# and Visual Basic .NET are essentially identical. There is a form builder onto which you drop controls. An object inspector allows you to set properties. You double-click on a control and a code window appears, already wired to the event. While the syntax of C# is considerably different than Visual Basic, many VB programmers will be pleasantly surprised at how similar the two are beneath the surface; over the years, VB has added support for most of the features in the Common Language Specification. But where some of these things have traditionally been optional, C# makes them mandatory.

In particular, in C# one must explicitly declare the types of all variables and, in general, is significantly more rigid about issues related to types. While this may seem an unnecessary burden at first, there are advantages to so-called *strong* typing. Probably the single most practical advantage is that when you explicitly spell out the type of the arguments and return values of a function (*method*, in C#'s preferred terminology), the Visual Studio .NET code editor can give you meaningful prompts about what type the values should be. As a practical matter,

this eliminates the problem of passing a string to a method that expects a number and vice versa.

Another difference that will strike Visual Basic programmers quickly is that C# has less whitespace than Visual Basic. Where a Visual Basic programmer might be used to typing **For Each**, in C# this will become **foreach**. Where Visual Basic's control structures use words to delimit scope (**If…Then…End If**and so forth), C#'s structures use curly brackets. This may strike some VB programmers as overly dense, but in time you will probably learn to prefer the quick single-keystroke of a curly bracket.

An enormous advantage of C# is that its syntax is derived from the C family of programming languages. This family of languages, when coupled with the **object.Field** referencing convention that is already familiar to Visual Basic programmers, absolutely dominates in the field of professional programming discourse. *Most* discussions of algorithms, program structure, and implementations are now written in this type of syntax; a C# programmer can participate in a much broader world of discussions than can a Visual Basic programmer, not because of any innate weaknesses in the VB syntax, but simply because a C#-like syntax has become the lingua franca of the programming community.

Just as importantly, the C# syntax prepares the Visual Basic programmer for the possibility of a significant career shift towards other programming languages such as C++ or Java. One of the sadder truths about programming as a career is that there is no such thing as job security. Many excellent programmers have found their careers ground on the rocks because the languages they have mastered have suddenly become obsolete. The speed with which languages fall out of favor is surprising; a few years ago, it was commonly believed that COBOL and dBase programmers would never have to worry about unemployment and Silicon Valley (an important bellwether of technology trends) has recently seen a precipitous drop in Java opportunities.[1] The two most important things in a programmer's career management are risk management and continual education. C# is an ideal risk management choice for VB programmers, as they risk nothing in terms of productivity, avoid the possibility of a decrease in demand for Visual Basic, and position themselves so that even if something catastrophic happens to the local demand for Windows programming, they know

[1] This is not to say that Java is in any danger of disappearing soon, as it is the most commonly taught language in colleges.

a C-derived language that gives them a "fighting chance" to move into another area of programming.

However, if C# is used as nothing but a C-like Visual Basic, programmers will have missed a golden opportunity to increase their productivity even beyond Visual Basic's capability. While Visual Basic can be said to be "object-based," most VB programmers use Forms as the central organizing principle of their solutions. With C# and "true" object-orientation, the graphical Form loses its central role and becomes just one more object among a number of others. The first *half* of this book is really an extended tutorial not so much on the C# language's syntax, but on object-oriented programming. OOP involves a very significant change in the way one *thinks* about programming; object orientation provides a way of envisioning, discussing, and making decisions about programs that most people agree is far more effective than the procedural way of thinking.

The first paragraphs of this discussion talked about how Visual Studio .NET provides a very familiar environment for Visual Basic programmers. However, we strongly advise readers, *especially* VB programmers, to avoid Visual Studio at least until Chapter 14's discussion of Windows Forms. One unfortunate thing about Visual Basic is that it fails to distinguish between the capabilities of the *language* and the capabilities of the *development environment*. The Form-centric architecture that characterizes many VB programs is largely due to the environment's behavior, not the inherent characteristics of the VB language. Because Visual Studio .NET provides an environment very similar to VB, it can actually stand in the way of "getting" OOP, which again is really the key to *thinking* in C#. By using a separate programming editor (or Visual Studio's editor, but without creating Visual Studio projects or solutions) and a command-line compiler, you will gain a much clearer sense of the essence of programming C#. Once you have learned C# to the point where you have no need for Visual Studio .NET, you'll discover that it provides a suite of tools that greatly speed certain tasks and you'll be free to choose to use those tools or not, making you *at least* as productive as you are today with Visual Basic, but with the huge added advantages that OOP brings to advanced software development.

B: C# for Java Programmers

Of course, no one really believes that Java's success has borne out its stated value proposition of "write once, run anywhere" (at least no one who believes that the Macintosh is a significant part of "anywhere").[1] Java has succeeded in two key areas: as the dominant language for writing server-side applications and as the top language for teaching computer science in colleges. The .NET Framework is better for both these areas, although obviously it is not inevitable that it will become dominant in either. For writing client applications, there is no question that C# clearly outstrips Java.

There are several good reasons for a Java programmer to learn C#. Before going into them, though, it's important to acknowledge that C# and .NET have been shaped in part by Sun's litigation over Microsoft's version of Java. Although Microsoft had been talking since at least 1991 about some C++-derived language to bridge the gap between Visual Basic and C++, Java's blistering pace of innovation in 1996 and 1997 gave Sun tremendous credibility regarding the claim that Java was commoditizing the operating system. By 1998, people (most importantly, Judge Ronald Whyte) were willing to believe that Microsoft's platform-specific variations were "an attempt to fragment the standardized application environment, and break with cross platform compatibility."[2] Even

---

[1] Yes, yes, the Macintosh supports Java and OS X brings the level of that support to a reasonable par with what's available on Windows and UNIX. But this is a recent occurrence and only became true long after Java had succeeded due to its other merits.

[2] Despite, incidentally, a license that said that Microsoft could "modify, adapt, and create Derivative Works of the Technology." Further, the finding that Microsoft's behavior would cause "irreparable harm" to Sun was not based on an analysis of the market for computer languages and technology, but on the finding that Microsoft's behavior constituted a

Bill Gates wrote in those days that "It's still very unclear to me what our OS will offer to Java client applications code that will make them unique enough to preserve our market position."

Now, it's much clearer. Java is just another platform, not the end of platforms. The pace of Java innovation has greatly slowed, especially when it comes to application-level features. It took years for Java to support mouse-wheels, Swing is not available on J2ME, J2EE is needlessly complex, and .NET leapfrogs Java in the area of Web Services.

To the extent that Java's goal was to commoditize the operating system, it has utterly failed. Windows is still important, as are the Macintosh operating systems, Linux, Solaris, and PalmOS. Obviously, a Java *programmer* has the advantage of being able to jump back and forth to develop solutions for any of these platforms, but Java does not come close to subsuming the capabilities of any operating system. If you're interested in providing your users a rich client-side experience, Java just isn't competitive with native applications. C# makes writing native Windows applications incredibly easy; Windows Forms is noticeably easier to program than Swing, has much better performance, a more comprehensive widget set, and COM Interop and PInvoke (technologies for accessing non-.NET resources) are vastly easier to use than Java Native Interfacing.

Obviously, if you choose to write a rich client application in C#, you're giving up on alternate operating systems, too. Most significantly, you're giving up on Macintosh and the PalmOS, which are the only two non-Windows platforms that are serious *markets* for client applications (Linux, which has a strong claim for being the most significant threat to Microsoft's OS dominance, is dominated by a mentality that thinks software should be free). However, there are only three significant areas where Microsoft does not yet provide a solution – mainframes, telephones, and the Macintosh. Mainframes are now legacy systems and Microsoft is poised to enter the telephone market, where it is likely to achieve the same type of success it has achieved in the handheld market (where PalmOS still has a marketshare lead, but Microsoft's PocketPC is an increasingly important player). Unfortunately, no one has announced any plans for a port of the Common Language Infrastructure to Macintosh (although Mono does run on Linux running on the PowerPC chip).

---

copyright violation rather than "just" a contract violation. Copyright violations carry with them the presumption of irreparable harm and thus do the wheels of justice turn when many billions of dollars are involved.

Aside from rich client applications, the .NET Framework SDK is an excellent programming platform for server-side programming. There is no separate programming model for enterprise development as there is with EJBs. While J2EE does provide for scalability, it does so by introducing considerable complexity and constraining the programmer in annoying ways. Just as Java was striking for how much easier it made programming network and simple graphical programs than using Microsoft Foundation Classes and MFC, so too is .NET striking for how much easier it makes backend programming than using J2EE. Although .NET is new, Microsoft really bought into the Web Services vision several years ago and Java is actually playing catch-up to .NET in terms of delivering value via XML and standard Web protocols.

Finally, .NET turns the value proposition of Java upside-down. Where Java proposes a single language to "write once, run anywhere," .NET proposes "Any language, one platform." For those who have a broad interest in programming, the dominance of object-oriented imperative programming has been a mixed blessing. On the one hand, it's the common ground from which the patterns movement sprang and the patterns movement was certainly among the most significant developments of the 1990s. On the other hand, it's introduced a certain blandness to an industry that used to support an "Exotic Language of the Month Club." One of the few non-environmental correlates to the highly variable measure of programmer productivity is the number of programming languages spoken; it is more significant than age, years of experience, or salary.

Different programming languages allow for much more dramatic differences in approach than different libraries. Java and C# are likely to produce similar structures to tackle similar problems; the chief differences would be the use of Java inner classes versus C#'s **delegate** types for event handling, C#'s lack of checked exceptions, and the exact names and methods in the library functions used. On the other hand, PROLOG (a declarative language) and C# would produce dramatically different structures to solve the same problem; declarative programming engages the mind in a different way than imperative programming. Java's position as the most-taught language in colleges is all well and good, but it necessarily limits the audience and constrains the topics of software development to those aspects of programming that Java embodies. The .NET Framework provides a much more robust infrastructure for teaching a much broader concept of programming to a much broader audience.

As a programming language, the Java developer will find in C# an evolutionary improvement, as a programming platform, .NET is superior to the Java standard edition and markedly superior to J2EE. For those Java programmers who have

become disillusioned with the evolution of Java and the behavior of Sun, C# provides a higher-productivity solution for very little investment in time.

C: Test-First Programming with NUnit

The number one way to decrease total development time is to increase the effort spent on quality. Throughout this book we have tried to emphasize the benefits of quality-oriented practices and in particular unit-testing, the testing of individual methods. Until recently, most programmers have not had access to a robust testing framework and have done unit-testing haphazardly. The JUnit testing framework, originally developed by Erich Gamma and Kent Beck for Java, changed that for Java programmers and has become an important tool in many Java teams. JUnit has been ported and extended in several different ways; Philip Craig initially ported it to .NET.

The .NET version of JUnit is called NUnit and is available at *http://www.nunit.org/*. As this book was nearing finalization, a significant update to NUnit was released. This version, NUnit 2.0, is much closer in spirit to a native .NET application and uses attributes in a way somewhat similar to what we suggested as a direction in Chapter 13. Contributions to this version were made by James Newkirk, Michael C. Two, Alexei A. Vorontsov, and Charlie Poole in addition to Philip Craig. NUnit is an excellent tool and should definitely be incorporated into your development process. To use NUnit, you must use the object-oriented and attribute facilities in C# and therefore this appendix assumes that you have made your way through the first two-thirds of this book.

To use NUnit, you inherit from the class **TestCase** and implement methods declared as **void Test*Xxx*( )**. Here's the simplest class possible:

```
//:AppendixC:Mine.cs
//Boiler plate - similar code in all NUnit test code
using NUnit.Framework;
using System;

public class Mine: TestCase {

   public Mine(String s) : base(s) {}
   protected static ITest Suite{
      get {return new TestSuite(typeof (Mine));}
   }
   //End of boiler plate. Write tests beginning here
   public void TestFramework(){
      Assertion.Assert(true);
   }
}///:~
```

Save this class as **Mine.cs** and compile it with a reference to the **NUnitCore.dll**:

```
csc /reference:"c:\program files\nunit
v2.0\bin\nunit.framework.dll" /target:library Mine.cs
```

This will compile the file into a .NET managed DLL called **Mine.dll.** Now, with **NUnit-Console.exe** in the path, run:

```
NUnit-Console /fixture:Mine /assembly:Mine.dll
```

To generate this output:

```
NUnit version 2.0.6
Copyright (C) 2002 James W. Newkirk, Michael C. Two, Alexei
A. Vorontsov.
Copyright (C) 2000-2002 Philip Craig.
All Rights Reserved.
.

Tests run: 1, Failures: 0, Not run: 0, Time: 0.050072
seconds.
```

The command line tells **NUnit-Console** to load the **TestFixture**-attributed class **Mine** from the assembly **Mine.DLL**. Then, **NUnit-Console** executes every **Test**-attributed method in that **TestFixture**-attributed class – in this case, that means **Mine.ATestMethod( )**. As long as no exception is propagated from the method, it is considered to be a success. NUnit provides a class called

Assertion that contains several static methods (**Assert( )**, **AssertNotNull( )**, **AssertEquals( )**, etc.) that throw an exception if their arguments do not evaluate to **true**. Add this method to the above class:

```
[Test] public void TestFailure(){
   Assertion.Assert("false is not true", false);
}
```

And recompile and rerun. Now, the output will be:

```
NUnit version 2.0.6
Copyright (C) 2002 James W. Newkirk, Michael C. Two, Alexei
A. Vorontsov.
Copyright (C) 2000-2002 Philip Craig.
All Rights Reserved.
..F

Tests run: 2, Failures: 1, Not run: 0, Time: 0.2203168
seconds

Failures:
1) Mine.TestFailure : false is not true
   at Mine.TestFailure()
```

Now, when NUnit-Console loads the class **Mine**, it discovers *two* test methods **Mine.ATestMethod( )** that continues to work fine, and **Mine.TestFailure( )** that generates a failure. The method that generated the failure is reported to the console, so that you can determine what caused the problem.

Of course, the two methods that we've shown aren't really testing anything. To be of any value, you run your own code to generate some kind of condition variable and then you test *that*.

For instance, the .NET Framework contains a **Random** class that has a static method **Random.Next(int max)** that is documented to return "a positive random number less than the specified maximum." Does it work? You might write this test method to find out:

```
[Test] public void TestRandomNext(){
   Random r = new Random();
   int MAX_PARAM = 10;
   int maxValue = 0;
   int minValue = 10000;
   for (int i = 0; i < 10000; i++){
```

```
      int aRandVal = r.Next(MAX_PARAM);
      if (aRandVal < minValue){
         minValue = aRandVal;
      }
      if (aRandVal > maxValue){
         maxValue = aRandVal;
      }
   }
   //"a positive integer…
   Assertion.Assert(minValue >= 0);
   //"… less than the specified maximum"
   Assertion.Assert(maxValue < MAX_PARAM);
}
```

(This puts aside the question of whether the values returned by the **Random** class have a biased distribution, but that would be a significantly more complex test case.)

Performance questions can be similarly tested using the .NET Framework's **DateTime** and **TimeSpan** classes. This example tests the ratio of time spent inserting 1, 10, and 100 million random values into a **Hashtable** with lots of collisions. Similar types of test can help flush out Big O problems in all sorts of algorithms.

```
//:appendixC:HashtableTest.cs
using System;
using System.Collections;
using NUnit.Framework;

[TestFixture] public class HashtableTest {
  [Test] public void TestInsertionStability(){
    double MAX_ACCEPTABLE_STABLE_DIFF = 1.0;
    Hashtable ht = new Hashtable();
    DateTime startOfSmallInsert = DateTime.Now;
    InsertRandomElements(ht, 1000000);
    TimeSpan smallInsertTime =
    DateTime.Now - startOfSmallInsert;
    DateTime startOfModerateInsert = DateTime.Now;
    InsertRandomElements(ht, 10000000);
    TimeSpan moderateInsertTime =
    DateTime.Now - startOfModerateInsert;
    DateTime startOfLargeInsert = DateTime.Now;
```

```
    InsertRandomElements(ht, 100000000);
    TimeSpan largeInsertTime =
    DateTime.Now - startOfLargeInsert;
    long smallTicks = smallInsertTime.Ticks;
    long medTicks = moderateInsertTime.Ticks;
    long longTicks = largeInsertTime.Ticks;

    Console.WriteLine("Raw times: {0} {1} {2}",
      smallInsertTime, moderateInsertTime,
      largeInsertTime);
    double medToSmallRatio =
    (double) medTicks / (double) smallTicks;
    double largeToMedRatio =
    (double) longTicks / (double) medTicks;
    double diffInRatios =
    largeToMedRatio - medToSmallRatio;
    Console.WriteLine("Difference in ratios: "
      + diffInRatios);
    Assertion.Assert(
      Math.Abs(diffInRatios)
      < MAX_ACCEPTABLE_STABLE_DIFF);
  }

  Random r = new Random();

  void InsertRandomElements(
      Hashtable map, int elementsToAdd){
    for (int i = 0; i < elementsToAdd; i++) {
      map[r.Next(100)] = r.Next(1000);
    }
  }
}///:~
```

In an agile development project, nothing is more important than obeying these rules: Write unit tests for everything that could fail, *before and during development* of the solution, and never check in code without running the entire suite of unit tests developed in the project. (There is a GUI version of NUnit, but the console version is easier to integrate into an automated build process.) It seems like a hassle to write testing code, but the counter-intuitive truth is that writing testing code absolutely, positively speeds up your code development. Once you work this way for two weeks, you will never look back.

So what do you do if the tests reveal a problem? This is where programmers skeptical of agile methods try to make their case. "You can't refactor your way out of a bad architecture," they'll say. And indeed, there's something to that. If you never test non-functional requirements such as performance until you've got tens of thousands of lines of source code and have across-the-board issues, it is unlikely a handful of changes will bring the application up to snuff. On the other hand, a comprehensive test suite built up of unit tests is of incredible benefit when you *do* need to make some sweeping change that touches every class in the system such as reimplementing a base class or changing a core interface.

The key to cutting this Gordian knot lies in the agile emphasis on small releases. When an XP programmer speaks of shortening release cycles, they aren't talking about cutting from an 18-month cycle to a 9-month cycle. They're talking about shortening it to *a handful of weeks*. It needs to be emphasized that this does not always mean a real deployment of the release to the entire customer base, but for Web Services programmers, for whom deployment is just a matter of the servers in the closet versus the servers in the collocation facility, you *should* deploy live every few weeks. This will seem insane until you have embraced unit-testing. You can deploy live every few weeks, because everything your application does is being validated several times per day and nothing is ever checked in that invalidates any of your application's behavior. (Or, if it does, the responsibility is instantly apportioned – "Joe checked in these classes and this test in this test class broke. Fix it Joe!")

Programmers of Web Services have it especially easy when it comes to embracing this model of development because Web Services are by definition "headless" – your responsibilities begin with accepting an XML request and end with returning XML. Writing NUnit unit tests that fully exercise a Web Service is, if not entirely trivial, largely mechanical, involving the transcription of an XML vertical industry standard into a suite of sample requests and responses. It is true that this will invariably flush out inconsistencies, gaps, and logic flaws in the industry standard, but such problems are the natural result of the lack of widespread XML experience. And, as an added bonus, a test suite that exercises the standard and embodies your interpretation of the standard's ambiguities is a huge asset in maintaining a relationship with the lurching pace of standards. If you do a decent job documenting your test cases and are willing to share, you have an excellent chance of gaining influence in the technical working groups. Naturally, since you are both defining the test cases and programming to match them, your company has the inside track on claiming compliance to the standard.

If a method or class is hard to test, your design is very likely to be flawed. Refactoring an implementation to increase testability almost always results in an overall improvement in design quality.

D: Programming the Pocket PC

One of the joys of working with C# is the ability to bring to bear the full power of modern programming tools on new hardware devices. As the desktop becomes a legacy formfactor, products running the Pocket and Tablet PC versions of Windows provide ample opportunity for new applications. C# is an ideal way to quickly bring such applications to market.

While the Tablet PC runs a modified version of the Windows XP operating system and therefore can take advantage of the full .NET Framework SDK as described throughout this book, and additionally has a managed library for interacting with "ink" – the fundamental data structure of pen input – the Pocket PC uses a slimmed down version of the .NET Framework SDK called the Compact .NET Framework.

Developing for the Compact .NET Framework is best done from inside Visual Studio, as the build-and-deploy process is more complicated than just compiling a **.cs** file. The "New Projects Wizard" in Visual Studio .NET 1.1 provides options for developing a "Smart Device Application." Following the Wizard, you will be given a choice for targeting Windows CE or Pocket PC. Windows CE is like the tool blocks by which one builds embedded operating systems and Pocket PC is one of the operating systems built with Windows CE. It's not exactly accurate to say that Pocket PC is built *on* Windows CE because Pocket PC is not a strict superset of Windows CE capabilities (Pocket PC does not support DCOM, for instance).

Once you've used the New Projects Wizard to generate the project file, working in Visual Studio .NET to develop a Pocket PC application is almost no different than developing a desktop application. The most significant difference is that the Compact .NET Framework does not support all the features of Windows Forms and GDI+. Laying out a GUI on Pocket PC typically involves more hard-coding of

positions and sizes than one would want to do in a desktop application, but this is a reasonable thing on a platform that has far less variation in screen resolutions.

This program is a cellular automata simulator for the Pocket PC. A one-dimensional cellular automaton is a simple array of cells that can be in one of two states. The state of a cell is determined by the state of itself and its immediate neighbors in the previous generation. For instance, you might have the rule "If the cell and its ancestors were on in the previous generation, the cell will be off in this generation." The state of every cell in the array is calculated this way and the generation is drawn on its own line on the screen. By repeatedly drawing generations this way, the evolution of the cellular automata over time can be read from the top of the screen to the bottom.

Although cellular automata (CAs) are simple to describe and program, they have surprising properties, notably, some CAs have been proven *computationally complete*, which is to say that given enough memory and time, these CAs can perform any calculation that is calculable by any computer. This program shows one such CA.

```
//:Appendix:CAPocketPC.cs
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using System.Threading;

namespace CAPocketPC
{
   public class Form1 : Form{
      Automata ca;
      Bitmap screenDisplay;
      Thread processThread;
      bool shouldRun = true;

      Form1(){
         ca = new Automata(this, this.Width, 110);
         screenDisplay =
         new Bitmap(this.Width, this.Height);
         Graphics g =
         Graphics.FromImage(screenDisplay);
         g.Clear(Color.Black);
```

```
      processThread =
      new Thread(new ThreadStart(ca.Run));
      processThread.Start();
   }

   public static void Main(string[] args){
      Application.Run(new Form1());
   }

   protected override void OnPaint(
      PaintEventArgs e){
      e.Graphics.DrawImage(screenDisplay, 0, 0);
   }

   public void NewGenerationReady(Automata a){
      OnNewGeneration();
      Invalidate();
   }

   int curLine = 0;

   private void OnNewGeneration(){
      Graphics g =
      Graphics.FromImage(screenDisplay);
      using(g){
         if (curLine == this.Height){
            g.Clear(Color.Black);
            curLine = 0;
         }
         Pen redPen = new Pen(Color.Red);
         Pen blackPen = new Pen(Color.Black);
         short[] pop = ca.Population;
         for (int i = 0; i < pop.Length; i++){
           short cell = pop[i];
           Pen pen =
               (cell == 0) ? redPen : blackPen;
           //Neat "shifting curtain" effect
           g.DrawRectangle(pen, i, curLine, i, 60);
         }
         curLine++;
      }
```

```
         Invalidate();
      }

      public void CloseHandler(
         object src, EventArgs e){
         shouldRun = false;
      }
   }

   public class Automata{
      bool shouldRun = true;
      public bool ShouldRun{
         set { shouldRun = value;}
         get { return shouldRun;}
      }
      static Rule[] RulesForDecimal(short rep){
         Rule[] rules = new Rule[8];
         int i = 0;
         for (short hiBit = 0; hiBit < 2; hiBit++){
           for
           (short medBit = 0; medBit < 2; medBit++){
             for
             (short lowBit = 0; lowBit < 2; lowBit++){
                short[] rule =
                  new short[]{hiBit, medBit, lowBit};
                short ruleBit = (short) (rep & 1);
                rules[i] = new Rule(rule, ruleBit, 1);
                Console.WriteLine(
hiBit + "" + medBit + "" + lowBit + "->" + ruleBit);
                i++;
                rep >>= 1;
              }
            }
         }
         return rules;
      }

      static Rule[] baseRules = new Rule[]{
         new Rule(new short[]{0, 1, 1,}, 1, 2),
         new Rule(new short[]{0, 0, 0, 0, 0}, 0, 2),
         new Rule(new short[]{0, 0, 0, 0, 0}, 0, 2),
```

```
        new Rule(new short[]{0, 0, 0, 0, 0}, 0, 2),
        new Rule(new short[]{0, 0, 0, 0, 0}, 0, 2),
    };

    int cellCount;
    short[] population;
    internal short[] Population{
        get{ return population;}
        set{ population = value;}
    }

    Rule[] rulebase;

    internal Automata(Form1 parentForm,
      int cellCount, short decRep):
      this(parentForm, cellCount,
           2, RulesForDecimal(decRep)){
    }

    internal Automata(Form1 parentForm,
      int cellCount):
      this(parentForm, cellCount, 2, baseRules) {
    }

    internal Automata(Form1 parentForm,
      int cellCount, int nStates, Rule[] rules){
       this.cellCount = cellCount;
       rulebase = rules;
       population = new short[cellCount];
       Random rand = new Random();
       for (int i = 0; i < population.Length; i++){
         population[i] = (short) rand.Next(nStates);
       }
       NewGeneration =
new AutomataHandler(parentForm.NewGenerationReady);
    }

    public void Run(){
       while (shouldRun)
          Generation();
    }
```

```
    short[] Generation(){
      short[] newGen = new short[population.Length];
      short width = rulebase[0].Width;
      short resultIndex = rulebase[0].ResultIndex;
      for (int i = 0; i < newGen.Length; i++){
          int pattern = 0;
          for
      (int ruleIdx = 0; ruleIdx < width; ruleIdx++){
             int prtIdx = i - resultIndex + ruleIdx;
             if (prtIdx < 0){
                prtIdx += population.Length;
             } else{
                if (prtIdx >= population.Length){
                   prtIdx -= population.Length;
                }
             }
             pattern += population[prtIdx]<<ruleIdx;
          }
          short result = rulebase[pattern].Result;
          newGen[(i + resultIndex) % cellCount]
             = result;
       }
       population = newGen;
       NewGeneration(this);
       return newGen;
    }

  public delegate void AutomataHandler(Automata ca);
  public event AutomataHandler NewGeneration;
}

class Rule{
   readonly public short ANY = -1;
   short[] predecessor;
   internal Rule(short[] predecessor, short result,
      short resultIndex){
      this.predecessor = predecessor;
      this.result = result;
      this.resultIdx = resultIndex;
      width = (short) predecessor.Length;
```

```
        }

        short width;
        public short Width{
           get{ return width;}
           set{width = value;}
        }

        short resultIdx;
        public short ResultIndex{
           get{ return resultIdx;}
           set{ resultIdx = value;}
        }

        short result;
        public short Result{
           get{ return result;}
           set{ result = value;}
        }

        public short this[int rulePos]{
           get{ return predecessor[rulePos];}
           set{ predecessor[rulePos] = value;}
        }
    }
}///:~
```

This code should be straightforward if you've read the chapters on GUI and multithreaded programming. The program works by creating an **Automata** of type "110." The numbering scheme is a shortcut way of referring to one of the 256 rulesets that can be generated for a
1-dimensional CA that has three ancestor cells that are in one of two states. The rule for an ancestor state of 000 is either 1 or 0, the rule for an ancestor state of 001 is either 1 or 0, etc. Once you've enumerated all the ancestor states, the rules for the CA is just an 8-digit sequence of 1s and 0. "110" is the decimal representation of a CA capable of universal computation ("30" is another interesting CA that generates a random sequence of very high quality).

After the ruleset for the **Automata** is created, the program starts a **Thread** tha calls **Automata.Run( )** that in turn calls **Automata.Generation( )**. That method consults the ruleset and the previous population and creates the new

population. Once the new population is ready, **Automata.Generation( )** triggers the **NewGeneration** event.

Form1.OnNewGeneration( ) responds to the **NewGeneration** event by drawing the **Automata**'s latest state into the **screenDisplay** bitmap. **Form1.OnPaint( )** simply paints the **screenDisplay** bitmap onto the screen.

As you can see, no special work needs to be done to accommodate the Pocket PC. The process of deploying to a real Pocket PC device or an emulator is taken care of by Visual Studio .NET.

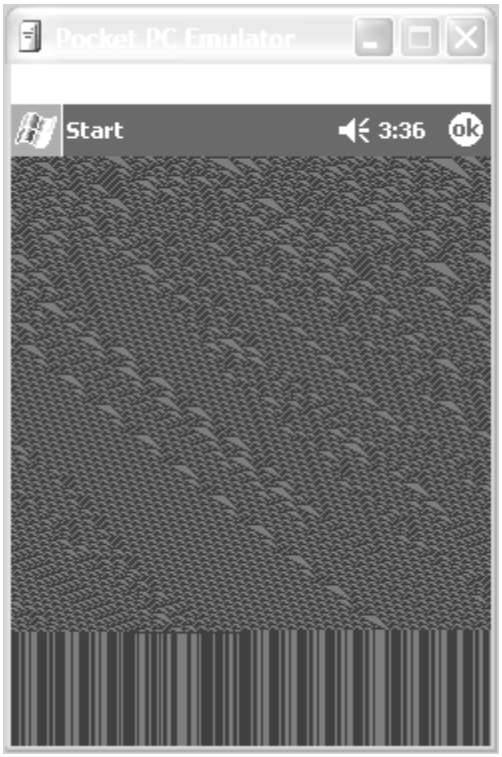


Figure D-1: A cellular automata running on a PocketPC emulator

When run, the 110 CA produces complex repetitive structures that can be seen as signals that propagate over time (remember that each line in the display represents a temporal step, so the bottom of the screen is a few hundred steps in time after the top). If given a large enough "working memory" (i.e., array width) and an appropriate starting state, the 110 CA can generate logic gates for the diagonally-propagating signals.

The dynamics of cellular automata are described in exhaustive detail in Stephen Wolfram's *A New Kind of Science* (Wolfram Media Inc.: 2002).

E: C# programming guidelines

This appendix contains suggestions to help guide you in performing low-level program design, and in writing code.

Naturally, these are guidelines and not rules. The idea is to use them as inspirations, and to remember that there are occasional situations where you need to bend or break a rule.

Design

1. **Deliver value**. It is the programmer's moral duty to make the life of the user better in some way. To do that, you must ship, and what you ship must be worthwhile to the end-user. Not to you, not to your peers, not to the marketing department, but to the user. This means writing and shipping programs that work as unobtrusively as possible to accomplish user goals in a manner that the user perceives as "natural." Theodore Sturgeon said "Sure, 90% of science fiction is crud. That's because 90% of everything is crud." Don't write crud. (And, yes, "crud" was the word he used.)

2. **Hurrying slows you down**. In the short term it might seem like it's faster to cut-and-paste a section of code that *almost* does what you want and modify it for a special case, or to skip writing a unit test because you need to get the functionality before you quit for the day, or to jump straight from a discussion with a customer to coding without writing down exactly what you think you've been asked to do. Haste doesn't work

in programming. If you want to experience profound leaps in productivity, you must strive for elegance, such that your solutions work the first time they are integrated and are easily modified in response to new user requests instead of requiring hours, days, or months of struggle. This point may take some experience to believe, because it can appear that you're not being productive while you're making a piece of code elegant. Elegance always pays off.

3. **First make it work, then make it fast**. This is true even if you are certain that a piece of code is really important and that it will be a principal bottleneck in your system. Don't do it. Get the system going first with as simple a design as possible. Then if it isn't going fast enough, profile it. You'll almost always discover that "your" bottleneck isn't the problem. Save your time for the really important stuff.

4. **Remember the "divide and conquer" principle**. If the problem you're looking at is too confusing, try to imagine what the basic operation of the program would be, given the existence of a magic "piece" that handles the hard parts. That "piece" is an object—write the code that uses the object, then look at the object and encapsulate *its* hard parts into other objects, etc.

13. **Separate the class creator from the class user (*client programmer*)**. The class user is the "customer" and doesn't need or want to know what's going on behind the scenes of the class. The class creator must be the expert in class design and write the class so that it can be used by the most novice programmer possible, yet still work robustly in the application. Library use will be easy only if it's transparent.

14. **When you create a class or method, attempt to make what it does so transparent that comments are unnecessary**. Your goal should be to make the client programmer's interface conceptually simple. To this end, use method overloading when appropriate to create an intuitive, easy-to-use interface.

15. **Your analysis and design must produce, at minimum, the classes in your system, their public interfaces, and their relationships to other classes, especially base classes**. If your design methodology produces more than that, ask yourself if all the pieces produced by that methodology have value over the lifetime of the program. If they do not, maintaining them will cost you. Members of

development teams tend not to maintain anything that does not contribute to their productivity; this is a fact of life that many design methods don't account for.

16. **Test everything**. Write the test code first (before you write the class), and keep it with the class. Automate the running of your tests through a makefile or similar tool. This way, any changes can be automatically verified by running the test code, and you'll immediately discover errors. Because you know that you have the safety net of your test framework, you will be bolder about making sweeping changes when you discover the need. Remember that the greatest improvements in languages come from the built-in testing provided by type checking, exception handling, etc., but those features take you only so far. You must go the rest of the way in creating a robust system by filling in the tests that verify features that are specific to your class or program.

17. **Write the test code first (before you write the class) in order to verify your class design**. If you can't write test code, you don't know what your class looks like. If a method is hard to test, it should probably be two or more methods. The act of writing the test code will often flush out additional features or constraints that you need in the class—these features or constraints don't always appear during analysis and design. Tests also provide example code showing how your class can be used.

18. **Automate build-and-test.** Using a makefile or similar tool, make it so that thoroughly testing your code is as simple as running a single command.

19. **All software design problems can be simplified by introducing an extra level of conceptual indirection**. Andrew Koenig has proposed this as a fundamental rule of software engineering. It is the basis of abstraction, the primary feature of object-oriented programming.

20. **An indirection should have a meaning** (in concert with guideline 9). This meaning can be something as simple as "putting commonly used code in a single method." If you add levels of indirection (abstraction, encapsulation, etc.) that don't have meaning, it can be as bad as not having adequate indirection.

21. **Make classes as atomic as possible**. Give each class a single, clear purpose. If your classes or your system design grows too complicated, break complex classes into simpler ones. The most obvious indicator of this is sheer size: if a class is big, chances are it's doing too much and

should be broken up.
Clues to suggest redesign of a class are:
1) A complicated switch statement: consider using polymorphism.
2) A large number of methods that cover broadly different types of operations: consider using several classes.
3) A large number of member variables that concern broadly different characteristics: consider using several classes.

22. **Watch for long argument lists**. Long argument lists are a symptom of a problem with coupling or cohesion (chapter 9). Such methods are difficult to write, read, and maintain. Instead, try to move the method to a class where it is (more) appropriate and refactor for better cohesion and less coupling.

23. **Don't repeat yourself**. If a piece of code recurs, put that code into a single method in the base class and call it from the derived-class methods. Not only do you save code space, you provide for easy propagation of changes. Sometimes the discovery of this common code will add valuable functionality to your interface.

24. **Watch for *switch* statements or chained *if-else* clauses**. This is typically an indicator of *type-check coding*, which means you are choosing what code to execute based on some kind of type information (the exact type may not be obvious at first). You can usually replace this kind of code with inheritance and polymorphism; a polymorphic method call will perform the type checking for you, and allow for more reliable and easier extensibility.

25. **From a design standpoint, look for and separate things that change from things that stay the same**. That is, search for the elements in a system that you might want to change without forcing a redesign, then encapsulate those elements in classes. You can learn significantly more about this concept in *Thinking in Patterns with Java*, downloadable at *www.BruceEckel.com*.

26. **Don't extend fundamental functionality by subclassing**. If an interface element is essential to a class it should be in the base class, not added during derivation. If you're adding methods by inheriting, perhaps you should rethink the design.

27. **Less is more**. Start with a minimal interface to a class, as small and simple as you need to solve the problem at hand, but don't try to anticipate all the ways that your class *might* be used. As the class is used,

you'll discover ways you must expand the interface. However, once a class is in use you cannot shrink the interface without disturbing client code. If you need to add more methods, that's fine; it won't disturb code, other than forcing recompiles. But even if new methods replace the functionality of old ones, leave the existing interface alone (you can combine the functionality in the underlying implementation if you want). If you need to expand the interface of an existing method by adding more arguments, create an overloaded method with the new arguments; this way you won't disturb any existing calls to the existing method.

28. **Read your class relationships aloud to make sure they're logical**. Refer to the relationship between a base class and derived class as "is-a" and member objects as "has-a."

29. **When deciding between inheritance and composition, ask if you need to upcast to the base type**. If not, prefer composition (member objects) to inheritance. This can eliminate the perceived need for multiple base types. If you inherit, users will think they are supposed to upcast.

30. **Use data members for variation in value and method overriding for variation in behavior**. That is, if you find a class that uses state variables along with methods that switch behavior based on those variables, you should probably redesign it to express the differences in behavior within subclasses and overridden methods.

31. **Watch for overloading**. A method should not conditionally execute code based on the value of an argument. In this case, you should create two or more overloaded methods instead.

32. **Use exception hierarchies**—preferably derived from specific appropriate classes in the standard C#Base Class Library exception hierarchy. The person catching the exceptions can then catch the specific types of exceptions, followed by the base type. If you add new derived exceptions, existing client code will still catch the exception through the base type.

33. **Sometimes simple aggregation does the job**. A "passenger comfort system" on an airline consists of disconnected elements: seat, air conditioning, video, etc., and yet you need to create many of these in a plane. Do you make private members and build a whole new interface? No—in this case, the components are also part of the public interface, so you should create public member objects. Those objects have their own

private implementations, which are still safe. Be aware that simple aggregation is not a solution to be used often, but it does happen.

34. **Consider the perspective of the client programmer and the person maintaining the code**. Design your class to be as obvious as possible to use. Anticipate the kind of changes that will be made, and design your class so that those changes will be easy.

35. **Watch out for "giant object syndrome."** This is often an affliction of procedural programmers who are new to OOP and who end up writing a procedural program and sticking it inside one or two giant objects. With the exception of application frameworks, objects represent concepts in your application, not the application.

36. **Always hide one thing.** Every type and method should encapsulate one (and only one) thing from the outside world. This doesn't mean that a class should just have one method! What is hidden varies between abstraction levels; classes operate at a much higher level of abstraction than methods. More challenging is that within a single class, it may be appropriate for different groups of methods to have different abstraction levels. For instance, the **private** methods of a class may hide one step of the more abstract "one thing" a **public** method hides.

37. **If you must do something ugly, make an abstraction for that service and localize it within a class**. This extra level of indirection prevents the ugly nature of what you're doing from being distributed throughout your program. (This idiom is embodied in the *Bridge* Pattern).

38. **OObjects should not simply hold some data**. They should also have well-defined behaviors. (If you need a "data object" to package and transport a group of items when a generalized container is innappropriate, use a **struct**.)

39. **Choose composition first when creating new classes from existing classes**. You should only used inheritance if it is required by your design. If you use inheritance where composition will work, your designs will become needlessly complicated.

40. **Use inheritance and method overriding to express differences in behavior, and fields to express variations in state**. An extreme example of what not to do is inheriting different classes to represent colors instead of using a "color" field.

41. **Watch out for *variance***. Two semantically different objects may have identical actions, or responsibilities, and there is a natural temptation to try to make one a subclass of the other just to benefit from inheritance. This is called variance, but there's no real justification to force a superclass/subclass relationship where it doesn't exist. A better solution is to create a general base class that produces an interface for both as derived classes—it requires a bit more space, but you still benefit from inheritance, and will probably make an important discovery about the design.

42. **Watch out for *limitation* during inheritance**. The clearest designs add new capabilities to inherited ones. A suspicious design removes old capabilities during inheritance without adding new ones. But rules are made to be broken, and if you are working from an old class library, it may be more efficient to restrict an existing class in its subclass than it would be to restructure the hierarchy so your new class fits in where it should, above the old class.

43. **Use design patterns to eliminate "naked functionality."** That is, if only one object of your class should be created, don't bolt ahead to the application and write a comment "Make only one of these." Wrap it in a singleton. If you have a lot of messy code in your main program that creates your objects, look for a creational pattern like a factory method in which you can encapsulate that creation. Eliminating "naked functionality" will not only make your code much easier to understand and maintain, it will also make it more bulletproof against the well-intentioned maintainers that come after you.

44. **Watch out for "analysis paralysis."** Remember that you must usually move forward in a project before you know everything, and that often the best and fastest way to learn about some of your unknown factors is to go to the next step rather than trying to figure it out in your head. You can't know the solution until you *have* the solution. Java C# has built-in firewalls; let them work for you. Your mistakes in a class or set of classes won't destroy the integrity of the whole system.

45. **Start with the simplest thing that could possibly work.** One important contributor to analysis paralysis is the introduction of variables for which you either have no data or are outside *today's* market. Will it work with quantum computers or people who have never used computers before? Unless quantum computers or computer illiterates have been identified *by your users and company* as important in the

near future, put aside your consideration of them. Don't build complexity into the system on the basis that it will simplify things later; develop habits and an environment that allow you to efficiently introduce complexity at the appropriate time.

46. **When you think you've got a good analysis, design, or implementation, do a walkthrough**. Bring someone in from outside your group—this doesn't have to be a consultant, but can be someone from another group within your company. Reviewing your work with a fresh pair of eyes can reveal problems at a stage when it's much easier to fix them, and more than pays for the time and money "lost" to the walkthrough process.

Implementation

47. **Whatever coding style you use, it really does make a difference if your team (and even better, your company) standardizes on it**. This means to the point that everyone considers it fair game to fix someone else's coding style if it doesn't conform. The value of standardization is that it takes less brain cycles to parse the code, so that you can focus more on what the code means.

48. **Follow standard capitalization rules**. Capitalize the first letter of class names and non-private methods and properties. The first letter of private fields, methods, and objects (references) should be lowercase. All identifiers should run their words together, and capitalize the first letter of all intermediate words. For example:
ThisIsAClassName
ThisIsANonPrivateMethodOrPropertyName
thisIsAPrivateMethodOrPropertyName
Capitalize *all* the letters of **readonly** values. This is a traditional way to indicate constants.

49. **Don't create your own "decorated" private data member names**. This is usually seen in the form of prepended underscores and characters. Hungarian notation is the worst example of this, where you attach extra characters that indicate data type, use, location, etc., as if you were writing assembly language and the compiler provided no extra assistance at all. These notations are confusing, difficult to read, and unpleasant to enforce and maintain. Let classes and packages do the name scoping for you. (Note that the .NET convention of prepending a capital **I** to interface names violates this rule.)

50. **Follow a "canonical form"** when creating a class for general-purpose use. Include definitions for **Equals( )**, **GetHashCode( )**, **ToString( )**, and **MemberwiseClone( ).** Implement **IDisposable** if the class contains non-memory resources that need to be explicitly finalized.

51. **For each non-private class, write a test class using NUnit or other unit-testing framework.** Test every non-private method that is not entirely trivial. Never check in code that breaks the test suite. You are not done with a task until you have a clean run through the test suite.

52. **Sometimes you need to inherit in order to access *protected* members of the base class**. This can lead to a perceived need for multiple base types. If you don't need to upcast, first derive a new class to perform the protected access. Then make that new class a member object inside any class that needs to use it, rather than inheriting.

53. If two classes are associated with each other in some functional way (such as containers and iterators), try to make one a nested class of the other. This not only emphasizes the association between the classes, but it allows the class name to be reused within a single package by nesting it within another class.

54. **Don't fall prey to premature optimization**. This way lies madness. In particular, don't worry about writing (or avoiding) native methods, making some methods non-virtual, or tweaking code to be efficient when you are first constructing the system. Your primary goal should be to deliver value to your clients and part of the discussion with them should be the non-functional performance characteristics they require.

55. **Keep scopes as small as possible so the visibility and lifetime of your objects are as small as possible**. This reduces the chance of using an object in the wrong context and hiding a difficult-to-find bug. For example, suppose you have a container and a piece of code that iterates through it. If you copy that code to use with a new container, you may accidentally end up using the size of the old container as the upper bound of the new one. If, however, the old container is out of scope, the error will be caught at compile-time.

56. **Use the containers in the .NET Framework SDK**. Become proficient with their use and you'll greatly increase your productivity.

57. **For a program to be robust, each component must be robust**. Use all the tools provided by C#: access control, exceptions, type

checking, and so on, in each class you create. That way you can safely move to the next level of abstraction when building your system.

58. **Prefer compile-time errors to run-time errors**. Try to handle an error as close to the point of its occurrence as possible. Prefer dealing with the error at that point to throwing an exception. Catch any exceptions in the nearest handler that has enough information to deal with them. Do what you can with the exception at the current level; if that doesn't solve the problem, rethrow the exception.

59. **Watch for long method definitions**. Methods should be brief, functional units that describe and implement a discrete part of a class interface. A method that is complicated is difficult and expensive to maintain, and is probably trying to do too much all by itself. If you see such a method, it indicates that, at the least, it should be broken up into multiple methods. It may also suggest the creation of a new class. Small methods will also foster reuse within your class. (The important thing is the complexity, not the number of lines of source code.)

60. **Keep things as "*private* as possible."** Once you publicize an aspect of your library (a method, a class, a field), you can never take it out. If you do, you'll wreck somebody's existing code, forcing them to rewrite and redesign. If you publicize only what you must, you can change everything else with impunity, and since designs tend to evolve this is an important freedom. In this way, implementation changes will have minimal impact on derived classes.

61. **Use comments liberally, and use the comment-documentation syntax to produce your program documentation**. However, the comments should add geniune meaning to the code; comments that only reiterate what the code is clearly expressing are annoying. Note that the typical verbose detail of C# class and method names reduce the need for as many comments.

62. **Avoid using "magic numbers"**—which are numbers hard-wired into code. These are a nightmare if you need to change them, since you never know if "100" means "the array size" or "something else entirely." Instead, create a constant with a descriptive name and use the constant identifier throughout your program. This makes the program easier to understand and much easier to maintain.

63. **When creating constructors, consider exceptions**. In the best case, the constructor won't do anything that throws an exception. In the

next-best scenario, the class will be composed and inherited from robust classes only, so they will need no cleanup if an exception is thrown. Otherwise, you must clean up composed classes inside a **finally** clause. If a constructor must fail, the appropriate action is to throw an exception, so the caller doesn't continue blindly, thinking that the object was created correctly.

64. **If your class requires any cleanup when the client programmer is finished with the object, make your class implement IDisposable.** Non-memory resources include handles to files or network streams, database connections, and references to non-managed objects. Remember that C#'s destructor is non-deterministic. However, if you implement **IDisposable**, you should also write a C# destructor that will serve as a "belts and suspender" last-chance to call **IDisposable.Dispose( )**. You should write your **Dispose( )** method so that it is robust even if called multiple times, but if your **Dispose( )** has a time-consuming implementation, use **GC.SuppressFinalize(this)** so that the destructor is not called.

65. **When you are creating a fixed-size container of objects, transfer them to an array**—especially if you're returning this container from a method. This way you get the benefit of the array's compile-time type checking, and the recipient of the array might not need to cast the objects in the array in order to use them.

66. **Choose *interfaces* over *abstract* classes**. If you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change it to an **abstract** class. An **interface** talks about what the client wants to do, while a class tends to focus on (or allow) implementation details.

67. **Inside constructors, do only what is necessary to set the object into the proper state**. Actively avoid calling other methods (except for **final** methods) since those methods can be overridden by someone else to produce unexpected results during construction. (See Chapter 5 for details.) Smaller, simpler constructors are less likely to throw exceptions or cause problems.

68. **Remember that code is read much more than it is written**. Clean designs make for easy-to-understand programs, but comments, detailed explanations, and examples are invaluable. They will help both you and

everyone who comes after you. If nothing else, the frustration of trying to ferret out useful information from the online Java documentation should convince you.

F: Resources

.NET Software

You do not need to buy Microsoft's excellent Visual Studio .NET Integrated Development Environment in order to program in C#. Every program in this book can be written, compiled, and run with a text editor and the command-line tools that are provided in Microsoft's .NET Platform SDK available at *http://msdn.microsoft.com*. It is a large download at 130MB, but is highly recommended.

Mono is an effort to make an Open Source implementation of the .NET Framework primarily targeting Linux and includes a free C# compiler. The Mono product is at *http://www.go-mono.com/*. Dot-GNU Portable .NET is a similar effort based in Britain and is available at *http://www.southern-storm.com.au/portable_net.html*.

SharpDevelop (*http://www.icsharpcode.net/opensource/sd/*) is an Open Source development environment for .NET written by Mike Kruger.

As discussed in Appendix C, NUnit is a unit testing framework by Philip Craig that is based on the JUnit framework written by Erich Gamma and Kent Beck. You can download NUnit from *http://www.nunit.org/*.

Non-.NET Books

Peopleware, by DeMarco and Lister (Dorset House, 1999). This is the most important book on software development ever published. It does not contain a line of code, but more than any other book it establishes that software development is not primarily a technical task at all, but success and productivity result from the psychic and physical environment in which development takes place.

Extreme Programming Explained by Beck (Addison Wesley, 2000), *Agile Modeling* by Ambler (Wiley, 2002), *The Unified Modeling Language User Guide* by Booch, Jacobson, and Rumbaugh (Addison Wesley, 1998). The furor over development methodologies is all about secondary considerations. "The driver of a software project is the customer. If the software doesn't do what they want it to do, you have failed." (Beck); "The primary goal of software development is to produce high-quality software that meets the needs of your project stakeholders

in an effective manner" (Ambler); "The primary product of a development team is not beautiful documents, world-class meetings, great slogans, or Pulitzer prize-winning lines of source code. Rather, it is good software that satisfies the evolving needs of its users and the business. Everything else is secondary" (Booch). Everyone agrees that the primary goal is delivering customer value and that the key to doing so effectively is *incremental and iterative* development. This is not to say that the debate about how much non-code artifacts contribute to this goal is not interesting and worthwhile and these three books effectively lay out the poles and a pragmatic middle ground.

Software for Use, by Constantine and Lockwood (Addison Wesley, 1999). While everyone agrees that customer value is what should drive the development process, *Software for Use* is the only book that actually describes effective techniques that place the user in control of what is developed. The book's heavy emphasis on user-interface issues follows from its premise that "to the user, the UI *is* the software."

Refactoring, by Fowler (Addison Wesley, 1999). This is the only book ever written on *rewriting* code. This is truly amazing in that everyone agrees that iteration is one of the key best practices in software development. Martin Fowler presents both a tutorial on when and how to go about restructuring existing code and an encyclopedia of "common refactorings" that are common to object-oriented code and therefore applicable to C#, although the book predates C#.

Software Assessments, Benchmarks, and Best Practices, by Jones (Addison Wesley, 2000). Even though most organizations have abysmal software development processes, they will not allow major changes unless a business case can be made that can show data relating to success and failure. This book is the best source of software productivity data, even if it is necessarily behind-the-times in terms of data associated with Web programming.

Concordance

:

:Add( )·413, 767, 808
:AddLine( )·679
:AppendChild( )·808
:AppendText( )·602
:BeginInit( )·699
:BeginInvoke( )·873
:BeginRead( )·861
:Close( )·465, 466, 861
:ConcreteMoveNext( )·410
:CreateDecryptor( )·484
:CreateEncryptor( )·484
:CreateGraphics( )·661, 712
:Dispose( )·172, 231, 232, 565, 700, 705
:Draw( )·36, 269
:EndInit( )·699
:EndInvoke( )·873
:EndRead( )·861
:Erase( )·36
:F( )·58
:Format( )·829
:GenerateIV( )·484
:GenerateKey( )·484
:GetItemCheckState( )·614
:GetLength( )·356, 357, 363, 364, 365, 369, 370, 373
:GetMethods( )·846
:GetProperties( )·846
:GetResourceSet( )·588
:GetResponseStream( )·860
:GetType( )·410, 480, 522, 593
:GetValues( )·419
:Highlight( )·811
:HorizontalTransform( )·364
:Interrupt( )·726, 862
:Invoke( )·873
:IsAssignableFrom( )·522
:IsMatch( )·500, 502, 507
:LoadFile( )·812
:Match( )·507
:Matches( )·502
:Navigate( )·700
:OnPaint( )·662, 666, 669, 672, 674, 676, 687, 688, 690, 693
:Play( )·272, 288
:Post( )·559
:println( )·240, 241
:ReadLine( )·466
:ReflectModel( )·580
:ResumeLayout( )·567
:Scrub( )·224
:SetItemCheckState( )·613
:SomeLogicalEvent( )·569
:Split( )·828
:Start( )·897
:SuspendLayout( )·566
:ToCharArray( )·476
:ToString( )·730, 767
:VerticalTransform( )·364
:Wash( )·244, 249, 250

B

BlockInfo: Build( )·827

C

CapStyle: Check( )·505; Close( )·505
Change: Act( )·295
Collections: ArrayList( )·203, 204
Columns: Add( )·422, 619, 620
Contains: BringToFront( )·642; Focus( )·642; SendToBack( )·642
Controller: AttachView( )·580
Controls: Add( )·571, 575, 576, 582, 583, 584, 585, 586, 593, 595, 600, 605, 609, 613, 616, 617, 620, 623, 630, 635, 637, 641, 642, 651, 660, 661, 663, 667, 670, 683, 684, 694, 696, 699, 705, 731, 743, 744, 810, 811; AddRange( )·567, 572, 575, 579, 588, 597, 601, 608, 611, 619, 631, 712, 733, 746; Contains( )·642; Remove( )·572, 705
Counter: Increment( )·746; ToString( )·188, 190
Current: Clone( )·805

D

Data: GetData( )·631, 632; GetDataPresent( )·631, 632
DataBindings: Add( )·637, 641
Debug: WriteLine( )·498
Decoder: GetChars( )·861
DomainSplitter: SplitString( )·571
Drawing: Point( )·566; Size( )·566, 596, 607, 712

E

Enum: GetValues( )·404, 411
Error: WriteLine( )·447, 449, 450, 452, 453, 454, 457, 465, 466, 467, 468, 469
EventLogTraceListener: Close( )·498

F

Forms: Button( )·566; Label( )·566

H

Handler: HandleRequest( )·554

I

Images: Add( )·598, 620
Int32: ToString( )·743
IntHolder: AddFan( )·282; AddPlayer( )·282
IsolatedStorageFile: GetUserStoreForAssembly( )·477
Items: Add( )·609, 613, 620; AddRange( )·610, 613; Contains( )·612

K

Key: CompareTo( )·419

L

Length: ToString( )·620
Links: Add( )·605
List: Add( )·400; Contains( )·400; CopyTo( )·400; IndexOf( )·400; Insert( )·400; Remove( )·400
Listeners: Add( )·498

M

Matrix: Clone( )·683
Measurement: Print( )·177
MenuItems: Add( )·648, 650, 651, 691, 694, 696, 811; AddRange( )·647, 648
Model: CloneMenu( )·648

N

Name: perhapsRelated( )·163; printGivenName( )·163
NameValueCollectionEntry: CloneMenu( )·648
Nodes: Add( )·617; Clear( )·810
NodeType: ToString( )·808

O

Out: Close( )·497; WriteLine( )·497

P

PDouble: Wash( )·243, 245
PictureBox: Invalidate( )·695, 697

R

RecursiveNotLocked: RecursiveCall( )·751
Relations: Add( )·817
Rows: Add( )·423, 425, 432

S

SByte: Parse( )·732, 745
SourceStream: ReadAll( )·481
Stack: Pop( )·382; Push( )·382
SubItems: Add( )·620
System: Activator; CreateInstance( )·519, 521, 524; Array; BinarySearch( )·357; Clear( )·357; Copy( )·356, 357; IndexOf( )·507, 580; Reverse( )·357, 476; Sort( )·357, 360, 361, 417, 420; Attribute; GetCustomAttribute( )·541; Console; OpenStandardOutput( )·799, 801; SetError( )·497; SetIn( )·497; SetOut( )·497; Write( )·108, 109, 359, 413, 423, 480, 484, 847; WriteLine( )·72, 77, 78, 83, 89, 90, 91, 93, 94, 95, 97, 98, 101, 102, 103, 104, 105, 108, 109, 117, 118, 129, 131, 132, 133, 134, 135, 136, 137, 142, 143, 145, 150, 151, 153, 154, 155, 157, 162, 163, 165, 168, 169, 171, 172, 174, 177, 180, 181, 182, 184, 185, 186, 187, 188, 191, 205, 207, 210, 211, 220, 221, 222, 223, 224, 226, 227, 228, 229, 230, 231, 232, 240, 242, 243, 244, 245, 249, 250, 252, 256, 262, 263, 264, 268, 271, 272, 273, 275, 276, 277, 279, 280, 282, 284, 288, 289, 290, 291, 293, 295, 303, 304, 306, 352, 353, 356, 357, 359, 361, 364, 369, 382, 383, 385, 386, 387, 388, 389,

390, 392, 393, 395, 396, 397, 398, 406, 408, 409, 410, 413, 414, 415, 416, 417, 420, 423, 426, 429, 432, 433, 445, 446, 447, 449, 454, 457, 458, 459, 460, 461, 462, 463, 464, 466, 474, 475, 477, 480, 485, 488, 492, 495, 496, 500, 502, 503, 506, 512, 515, 516, 520, 521, 524, 529, 530, 531, 534, 535, 536, 539, 540, 541, 548, 549, 552, 554, 555, 558, 559, 560, 574, 608, 609, 610, 611, 612, 614, 617, 625, 626, 628, 642, 651, 662, 666, 687, 695, 697, 700, 703, 704, 716, 717, 718, 719, 720, 725, 726, 728, 729, 748, 749, 750, 751, 752, 753, 766, 775, 776, 777, 789, 793, 799, 803, 805, 809, 810, 825, 828, 831, 833, 834, 836, 838, 839, 840, 841, 842, 843, 844, 846, 847, 850, 853, 860, 861, 862, 874, 898; DateTime; Parse( )·829; ToString( )·874; EventHandler( )·566, 568, 574; GC; SuppressFinalize( )·172; Guid; NewGuid( )·792; IDisposable; Dispose( )·461; Math; Cos( )·678, 684; Floor( )·363; Sin( )·663, 667, 678, 684; Sqrt( )·440; Object; GetType( )·520, 521, 524; Print( )·403; Random; Next( )·91, 93, 94, 99, 103, 107, 108, 145, 187, 188, 191, 193, 269, 360, 361, 385, 386, 387, 413, 414, 415, 416, 419, 519, 521, 578, 602, 603, 697, 705, 713, 890, 899; NextDouble( )·99, 133, 142, 143, 364, 454, 554, 578, 602, 726; String; Format( )·190; Type; GetConstructor( )·541; GetInterfaces( )·524; GetMethods( )·541; GetType( )·516, 519, 524; IsSubclassOf( )·521, 847

System.Collections: ArrayList; Add( )·203, 385, 398, 402, 411, 519, 521, 705; BinarySearch( )·414; Contains( )·411; FixedSize( )·385; GetEnumerator( )·402; ReadOnly( )·385; Remove( )·705; Reverse( )·414; Sort( )·414; BitArray; Invalidate( )·671; Set( )·386, 387; Hashtable; Add( )·389; ICollection; CopyTo( )·417; IDictionary; Add( )·390, 416; CopyTo( )·416; IEnumerator; MoveNext( )·513; IList; Add( )·414, 513; CopyTo( )·415; GetEnumerator( )·513; Queue; Dequeue( )·382; Enqueue( )·382; SortedList; Add( )·391, 392; GetByIndex( )·392; Synchronized( )·765

System.Collections.Specialized: NameValueCollection; Add( )·393, 419; GetValues( )·393; StringDictionary; ContainsKey( )·396

System.Data: DataRow; BeginEdit( )·432; Delete( )·433; EndEdit( )·432; DataSet; ReadXml( )·801; WriteXml( )·799, 816; WriteXmlSchema( )·799, 801; DataTable; NewRow( )·422, 423; IDataAdapter; Fill( )·427, 432, 635, 638, 643, 799, 806; Update( )·432, 433, 643; IDataReader; Close( )·429; Read( )·429; IDbCommand; ExecuteReader( )·429

System.Data.OleDb: OleDbConnection; Close( )·427, 429, 799, 806, 817; Open( )·427, 429, 432, 635, 638, 643, 799, 805, 816; OleDbDataAdapter; Fill( )·816

System.Diagnostics: Process; Start( )·605

System.Drawing: Bitmap; SetPixel( )·697; Color; Dispose( )·700, 705; FromArgb( )·674, 697; Graphics; BeginContainer( )·680; Clear( )·661, 662, 666, 669, 682, 704, 896, 897; DrawCurve( )·669; DrawEllipse( )·689; DrawImage( )·693, 897; DrawLine( )·667, 672; DrawPath( )·680, 687; DrawRectangle( )·663, 672, 682, 690, 693, 713, 897; DrawString( )·680, 681, 689, 690, 692; EndContainer( )·680; FillRectangle( )·672, 674, 675, 676, 695; FromImage( )·695, 896, 897; GetHdc( )·704; MeasureString( )·688; ReleaseHdc( )·704; RotateTransform( )·669, 680, 690; ScaleTransform( )·666, 669, 680, 690; TranslateTransform( )·666, 669, 680; Image; FromFile( )·586, 620, 674, 688, 695; Save( )·695

System.Drawing.Drawing2D: GraphicsPath; AddCurve( )·687; AddEllipse( )·677; AddLine( )·676, 687; CloseFigure( )·687; IsVisible( )·687

System.IO: BinaryReader; ReadBoolean( )·488; ReadByte( )·488; ReadBytes( )·488; ReadChar( )·488; ReadChars( )·488; ReadDecimal( )·488; ReadDouble( )·488; ReadString( )·488; BinaryWriter; Write( )·486, 487; Directory; CreateDirectory( )·476; Delete( )·476; Exists( )·476; GetCurrentDirectory( )·475; GetDirectories( )·475; GetFiles( )·474, 502; File; Create( )·476; OpenText( )·827; FileStream; Close( )·484, 485, 495, 592, 789, 816; Seek( )·495; WriteByte( )·484; Stream; Close( )·486, 487, 488, 491, 492, 506; CreateNavigator( )·808; ReadByte( )·495; Seek( )·494, 495; StreamReader; Close( )·501, 502, 504, 828; ReadLine( )·831, 834, 836, 839, 841; ReadToEnd( )·793, 853; StreamWriter; Close( )·493; Flush( )·831; WriteLine( )·492, 831, 834, 836, 839, 841

System.Net: Dns; GetHostByName( )·825; Resolve( )·832, 836, 840; HttpWebRequest; GetResponse( )·853; HttpWebResponse; GetResponseStream( )·853; IPAddress; Parse( )·829; WebRequest; Create( )·853, 860

System.Net.Sockets: TcpClient; Close( )·831, 834, 836, 839, 841; Connect( )·836, 841; GetStream( )·831, 833, 836, 839, 841; TcpListener; AcceptTcpClient( )·833, 838; Start( )·833, 838; Stop( )·834, 838

System.Reflection: Assembly; GetAssembly( )·534; GetExportedTypes( )·846; GetName( )·534; GetType( )·541, 846; Load( )·541, 846

System.Resources: ResourceManager; GetObject( )·593; ResourceSet; GetString( )·588; ResourceWriter; AddResource( )·592; Generate( )·592
System.Text: StringBuilder; Append( )·831, 843, 861; ToString( )·831, 861; UnicodeEncoding; GetBytes( )·481
System.Text.RegularExpressions: Regex; Match( )·540; Replace( )·503; Split( )·570
System.Threading: Interlocked; Decrement( )·704; Increment( )·703; Monitor; Enter( )·736, 741, *769*; Exit( )·736, 741; Mutex; WaitOne( )·752, 753
System.Web: HttpRequest; BeginGetResponse( )·860; EndGetResponse( )·860; HttpResponse; Write( )·855
System.Windows.Forms: Application; Exit( )·812; Run( )·562, 567, 572, 576, 580, 582, 583, 584, 585, 586, 588, 593, 595, 597, 600, 603, 605, 609, 610, 612, 614, 616, 617, 621, 624, 632, 635, 638, 643, 649, 652, 661, 663, 667, 671, 673, 674, 675, 676, 677, 684, 687, 689, 690, 691, 693, 695, 697, 700, 705, 713, 732, 745, 812, 897; Clipboard; GetDataObject( )·623; SetDataObject( )·623, 631; ColorDialog; ShowDialog( )·651; DataGrid; SetDataBinding( )·635; FontDialog; ShowDialog( )·652; IDataObject; GetData( )·623; GetDataPresent( )·623; Label; Focus( )·642; MessageBox; Show( )·605; OpenFileDialog; ShowDialog( )·651, 695, 697, 812; PrintDialog; ShowDialog( )·691; PrintPreviewDialog; ShowDialog( )·691; SaveFileDialog; ShowDialog( )·651, 695; Screen; GetBounds( )·712
System.Xml: XmlDataDocument; CreateNavigator( )·805, 809; XmlDocument; CreateNavigator( )·803; Load( )·777, 803, 810; XmlTextReader; Close( )·776; MoveToNextAttribute( )·776; Read( )·775
System.Xml.Serialization: XmlSerializer; Deserialize( )·789, 793; Serialize( )·784, 786, 791, 793
System.Xml.Xpath: XPathNodeIterator; MoveNext( )·803, 805, 809
System.Xml.Xsl: XslTransform; Load( )·820; Transform( )·820

T

Tables: Add( )·422
Thread: Abort( )·733, 746, 747, 767; Join( )·728, 729; Sleep( )·168, 172, 252, 713, 718, 725, 726, 729, 733, 734, 746, 747, 748, 750, 753, 766, 850, 862; Start( )·716, 718, 720, 727, 728, 729, 733, 746, 747, 748, 749, 767, 838, 840
ToolTip: SetToolTip( )·600
Trace: WriteLine( )·498
Transaction: Process( )·454
Tree: Info( )·153
TwoCounter: IncrementAccess( )·731, 743

U

Useful: Fight( )·306; Fly( )·306; Swim( )·306
UtilityCo: BeginBilling( )·559

W

WhatsTheTime: Time( )·874
WriteLine: play( )·237; ToString( )·836, 840

Class, Method, Property Cross-Reference

•

.Add( ),413, 767, 808
.AddLine( ),679
.AppendChild( ),808
.AppendText( ),602
.BeginInit( ),699
.BeginInvoke( ),873
.BeginRead( ),861
.Close( ),465, 466, 861
.ConcreteMoveNext( ),410
.CreateDecryptor( ),484
.CreateEncryptor( ),484
.CreateGraphics( ),661, 712
.Dispose( ),172, 231, 232, 565, 700, 705
.Draw( ),36, 269
.EndInit( ),699
.EndInvoke( ),873
.EndRead( ),861
.Erase( ),36
.F( ),58
.Format( ),829
.GenerateIV( ),484
.GenerateKey( ),484
.GetItemCheckState( ),614
.GetLength( ),356, 357, 363, 364, 365, 369, 370, 373
.GetMethods( ),846
.GetProperties( ),846
.GetResourceSet( ),588
.GetResponseStream( ),860
.GetType( ),410, 480, 522, 593
.GetValues( ),419
.Highlight( ),811
.HorizontalTransform( ),364
.Interrupt( ),726, 862
.Invoke( ),873
.IsAssignableFrom( ),522
.IsMatch( ),500, 502, 507
.LoadFile( ),812
.Match( ),507
.Matches( ),502
.Navigate( ),700
.OnPaint( ),662, 666, 669, 672, 674, 676, 687, 688, 690, 693
.Play( ),272, 288
.Post( ),559
.println( ),240, 241
.ReadLine( ),466
.ReflectModel( ),580
.ResumeLayout( ),567
.Scrub( ),224
.SetItemCheckState( ),613
.SomeLogicalEvent( ),569
.Split( ),828
.Start( ),897
.SuspendLayout( ),566
.ToCharArray( ),476
.ToString( ),730, 767
.VerticalTransform( ),364
.Wash( ),244, 249, 250

A

Abort( ): in Thread,733, 746, 747, 767
AcceptTcpClient( ): in TcpListener,833, 838
Act( ): in Change,295
Activator.CreateInstance( ),519, 521, 524
Add( ): in,413, 767, 808; in ArrayList,203, 385, 398, 402, 411, 519, 521, 705; in Columns,422, 619, 620; in Controls,571, 575, 576, 582, 583, 584, 585, 586, 593, 595, 600, 605, 609, 613, 616, 617, 620, 623, 630, 635, 637, 641, 642, 651, 660, 661, 663, 667, 670, 683, 684, 694, 696, 699, 705, 731, 743, 744, 810, 811; in DataBindings,637, 641; in Hashtable,389; in IDictionary,390, 416; in IList,414, 513; in Images,598, 620; in Items,609, 613, 620; in Links,605; in List,400; in Listeners,498; in MenuItems,648, 650, 651, 691, 694, 696,

811; in NameValueCollection,393, 419; in Nodes,617; in Relations,817; in Rows,423, 425, 432; in SortedList,391, 392; in SubItems,620; in Tables,422
AddCurve( ): in GraphicsPath,687
AddEllipse( ): in GraphicsPath,677
AddFan( ): in IntHolder,282
AddLine( ): in,679; in GraphicsPath,676, 687
AddPlayer( ): in IntHolder,282
AddRange( ): in Controls,567, 572, 575, 579, 588, 597, 601, 608, 611, 619, 631, 712, 733, 746; in Items,610, 613; in MenuItems,647, 648
AddResource( ): in ResourceWriter,592
Append( ): in StringBuilder,831, 843, 861
AppendChild( ): in,808
AppendText( ): in,602
Application.Exit( ),812
Application.Run( ),562, 567, 572, 576, 580, 582, 583, 584, 585, 586, 588, 593, 595, 597, 600, 603, 605, 609, 610, 612, 614, 616, 617, 621, 624, 632, 635, 638, 643, 649, 652, 661, 663, 667, 671, 673, 674, 675, 676, 677, 684, 687, 689, 690, 691, 693, 695, 697, 700, 705, 713, 732, 745, 812, 897
Array.BinarySearch( ),357
Array.Clear( ),357
Array.Copy( ),356, 357
Array.IndexOf( ),507, 580
Array.Reverse( ),357, 476
Array.Sort( ),357, 360, 361, 417, 420
ArrayList( ): in Collections,203, 204
ArrayList.Add( ),203, 385, 398, 402, 411, 519, 521, 705
ArrayList.BinarySearch( ),414
ArrayList.Contains( ),411
ArrayList.FixedSize( ),385
ArrayList.GetEnumerator( ),402
ArrayList.ReadOnly( ),385
ArrayList.Remove( ),705
ArrayList.Reverse( ),414
ArrayList.Sort( ),414
Assembly.GetAssembly( ),534
Assembly.GetExportedTypes( ),846
Assembly.GetName( ),534
Assembly.GetType( ),541, 846
Assembly.Load( ),541, 846
AttachView( ): in Controller,580
Attribute.GetCustomAttribute( ),541

B

BeginBilling( ): in UtilityCo,559
BeginContainer( ): in Graphics,680
BeginEdit( ): in DataRow,432
BeginGetResponse( ): in HttpRequest,860
BeginInit( ): in,699
BeginInvoke( ): in,873
BeginRead( ): in,861
BinaryReader.ReadBoolean( ),488
BinaryReader.ReadByte( ),488
BinaryReader.ReadBytes( ),488
BinaryReader.ReadChar( ),488
BinaryReader.ReadChars( ),488
BinaryReader.ReadDecimal( ),488
BinaryReader.ReadDouble( ),488
BinaryReader.ReadString( ),488
BinarySearch( ): in Array,357; in ArrayList,414
BinaryWriter.Write( ),486, 487
BitArray.Invalidate( ),671
BitArray.Set( ),386, 387
Bitmap.SetPixel( ),697
BlockInfo.Build( ),827
BringToFront( ): in Contains,642
Build( ): in BlockInfo,827
Button( ): in Forms,566

C

CapStyle.Check( ),505
CapStyle.Close( ),505
Change.Act( ),295
Check( ): in CapStyle,505
Clear( ): in Array,357; in Graphics,661, 662, 666, 669, 682, 704, 896, 897; in Nodes,810
Clipboard.GetDataObject( ),623
Clipboard.SetDataObject( ),623, 631
Clone( ): in Current,805; in Matrix,683
CloneMenu( ): in Model,648; in NameValueCollectionEntry,648
Close( ): in,465, 466, 861; in CapStyle,505; in EventLogTraceListener,498; in FileStream,484, 485, 495, 592, 789, 816; in IDataReader,429; in OleDbConnection,427, 429, 799, 806, 817; in Out,497; in Stream,486, 487, 488, 491, 492, 506; in StreamReader,501, 502, 504, 828; in StreamWriter,493; in TcpClient,831, 834, 836, 839, 841; in XmlTextReader,776
CloseFigure( ): in GraphicsPath,687
Collections.ArrayList( ),203, 204
Color.Dispose( ),700, 705
Color.FromArgb( ),674, 697
ColorDialog.ShowDialog( ),651
Columns.Add( ),422, 619, 620
CompareTo( ): in Key,419
ConcreteMoveNext( ): in,410
Connect( ): in TcpClient,836, 841
Console.OpenStandardOutput( ),799, 801
Console.SetError( ),497
Console.SetIn( ),497
Console.SetOut( ),497
Console.Write( ),108, 109, 359, 413, 423, 480, 484, 847

Console.WriteLine( ),72, 77, 78, 83, 89, 90, 91, 93, 94, 95, 97, 98, 101, 102, 103, 104, 105, 108, 109, 117, 118, 129, 131, 132, 133, 134, 135, 136, 137, 142, 143, 145, 150, 151, 153, 154, 155, 157, 162, 163, 165, 168, 169, 171, 172, 174, 177, 180, 181, 182, 184, 185, 186, 187, 188, 191, 205, 207, 210, 211, 220, 221, 222, 223, 224, 226, 227, 228, 229, 230, 231, 232, 240, 242, 243, 244, 245, 249, 250, 252, 256, 262, 263, 264, 268, 271, 272, 273, 275, 276, 277, 279, 280, 282, 284, 288, 289, 290, 291, 293, 295, 303, 304, 306, 352, 353, 356, 357, 359, 361, 364, 369, 382, 383, 385, 386, 387, 388, 389, 390, 392, 393, 395, 396, 397, 398, 406, 408, 409, 410, 413, 414, 415, 416, 417, 420, 423, 426, 429, 432, 433, 445, 446, 447, 449, 454, 457, 458, 459, 460, 461, 462, 463, 464, 466, 474, 475, 477, 480, 485, 488, 492, 495, 496, 500, 502, 503, 506, 512, 515, 516, 520, 521, 524, 529, 530, 531, 534, 535, 536, 539, 540, 541, 548, 549, 552, 554, 555, 558, 559, 560, 574, 608, 609, 610, 611, 612, 614, 617, 625, 626, 628, 642, 651, 662, 666, 687, 695, 697, 700, 703, 704, 716, 717, 718, 719, 720, 725, 726, 728, 729, 748, 749, 750, 751, 752, 753, 766, 775, 776, 777, 789, 793, 799, 803, 805, 809, 810, 825, 828, 831, 833, 834, 836, 838, 839, 840, 841, 842, 843, 844, 846, 847, 850, 853, 860, 861, 862, 874, 898
Contains( ): in ArrayList,411; in Controls,642; in Items,612; in List,400
Contains.BringToFront( ),642
Contains.Focus( ),642
Contains.SendToBack( ),642
ContainsKey( ): in StringDictionary,396
Controller.AttachView( ),580
Controls.Add( ),571, 575, 576, 582, 583, 584, 585, 586, 593, 595, 600, 605, 609, 613, 616, 617, 620, 623, 630, 635, 637, 641, 642, 651, 660, 661, 663, 667, 670, 683, 684, 694, 696, 699, 705, 731, 743, 744, 810, 811
Controls.AddRange( ),567, 572, 575, 579, 588, 597, 601, 608, 611, 619, 631, 712, 733, 746
Controls.Contains( ),642
Controls.Remove( ),572, 705
Copy( ): in Array,356, 357
CopyTo( ): in ICollection,417; in IDictionary,416; in IList,415; in List,400
Cos( ): in Math,678, 684
Counter.Increment( ),746
Counter.ToString( ),188, 190
Create( ): in File,476; in WebRequest,853, 860
CreateDecryptor( ): in,484
CreateDirectory( ): in Directory,476
CreateEncryptor( ): in,484
CreateGraphics( ): in,661, 712
CreateInstance( ): in Activator,519, 521, 524
CreateNavigator( ): in Stream,808; in XmlDataDocument,805, 809; in XmlDocument,803
Current.Clone( ),805

D

Data.GetData( ),631, 632
Data.GetDataPresent( ),631, 632
DataBindings.Add( ),637, 641
DataGrid.SetDataBinding( ),635
DataRow.BeginEdit( ),432
DataRow.Delete( ),433
DataRow.EndEdit( ),432
DataSet.ReadXml( ),801
DataSet.WriteXml( ),799, 816
DataSet.WriteXmlSchema( ),799, 801
DataTable.NewRow( ),422, 423
DateTime.Parse( ),829
DateTime.ToString( ),874
Debug.WriteLine( ),498
Decoder.GetChars( ),861
Decrement( ): in Interlocked,704
Delete( ): in DataRow,433; in Directory,476
Dequeue( ): in Queue,382
Deserialize( ): in XmlSerializer,789, 793
Directory.CreateDirectory( ),476
Directory.Delete( ),476
Directory.Exists( ),476
Directory.GetCurrentDirectory( ),475
Directory.GetDirectories( ),475
Directory.GetFiles( ),474, 502
Dispose( ): in,172, 231, 232, 565, 700, 705; in Color,700, 705; in IDisposable,461
Dns.GetHostByName( ),825
Dns.Resolve( ),832, 836, 840
DomainSplitter.SplitString( ),571
Draw( ): in,36, 269
DrawCurve( ): in Graphics,669
DrawEllipse( ): in Graphics,689
DrawImage( ): in Graphics,693, 897
Drawing.Point( ),566
Drawing.Size( ),566, 596, 607, 712
DrawLine( ): in Graphics,667, 672
DrawPath( ): in Graphics,680, 687
DrawRectangle( ): in Graphics,663, 672, 682, 690, 693, 713, 897
DrawString( ): in Graphics,680, 681, 689, 690, 692

E

EndContainer( ): in Graphics,680
EndEdit( ): in DataRow,432
EndGetResponse( ): in HttpRequest,860

EndInit( ): in,699
EndInvoke( ): in,873
EndRead( ): in,861
Enqueue( ): in Queue,382
Enter( ): in Monitor,736, 741, *769*
Enum.GetValues( ),404, 411
Erase( ): in,36
Error.WriteLine( ),447, 449, 450, 452, 453, 454, 457, 465, 466, 467, 468, 469
EventHandler( ): in System,566, 568, 574
EventLogTraceListener.Close( ),498
ExecuteReader( ): in IDbCommand,429
Exists( ): in Directory,476
Exit( ): in Application,812; in Monitor,736, 741

F

F( ): in,58
Fight( ): in Useful,306
File.Create( ),476
File.OpenText( ),827
FileStream.Close( ),484, 485, 495, 592, 789, 816
FileStream.Seek( ),495
FileStream.WriteByte( ),484
Fill( ): in IDataAdapter,427, 432, 635, 638, 643, 799, 806; in OleDbDataAdapter,816
FillRectangle( ): in Graphics,672, 674, 675, 676, 695
FixedSize( ): in ArrayList,385
Floor( ): in Math,363
Flush( ): in StreamWriter,831
Fly( ): in Useful,306
Focus( ): in Contains,642; in Label,642
FontDialog.ShowDialog( ),652
Format( ): in,829; in String,190
Forms.Button( ),566
Forms.Label( ),566
FromArgb( ): in Color,674, 697
FromFile( ): in Image,586, 620, 674, 688, 695
FromImage( ): in Graphics,695, 896, 897

G

GC.SuppressFinalize( ),172
Generate( ): in ResourceWriter,592
GenerateIV( ): in,484
GenerateKey( ): in,484
GetAssembly( ): in Assembly,534
GetBounds( ): in Screen,712
GetByIndex( ): in SortedList,392
GetBytes( ): in UnicodeEncoding,481
GetChars( ): in Decoder,861
GetConstructor( ): in Type,541
GetCurrentDirectory( ): in Directory,475
GetCustomAttribute( ): in Attribute,541
GetData( ): in Data,631, 632; in IDataObject,623
GetDataObject( ): in Clipboard,623
GetDataPresent( ): in Data,631, 632; in IDataObject,623
GetDirectories( ): in Directory,475
GetEnumerator( ): in ArrayList,402; in IList,513
GetExportedTypes( ): in Assembly,846
GetFiles( ): in Directory,474, 502
GetHdc( ): in Graphics,704
GetHostByName( ): in Dns,825
GetInterfaces( ): in Type,524
GetItemCheckState( ): in,614
GetLength( ): in,356, 357, 363, 364, 365, 369, 370, 373
GetMethods( ): in,846; in Type,541
GetName( ): in Assembly,534
GetObject( ): in ResourceManager,593
GetProperties( ): in,846
GetResourceSet( ): in,588
GetResponse( ): in HttpWebRequest,853
GetResponseStream( ): in,860; in HttpWebResponse,853
GetStream( ): in TcpClient,831, 833, 836, 839, 841
GetString( ): in ResourceSet,588
GetType( ): in,410, 480, 522, 593; in Assembly,541, 846; in Object,520, 521, 524; in Type,516, 519, 524
GetUserStoreForAssembly( ): in IsolatedStorageFile,477
GetValues( ): in,419; in Enum,404, 411; in NameValueCollection,393
Graphics.BeginContainer( ),680
Graphics.Clear( ),661, 662, 666, 669, 682, 704, 896, 897
Graphics.DrawCurve( ),669
Graphics.DrawEllipse( ),689
Graphics.DrawImage( ),693, 897
Graphics.DrawLine( ),667, 672
Graphics.DrawPath( ),680, 687
Graphics.DrawRectangle( ),663, 672, 682, 690, 693, 713, 897
Graphics.DrawString( ),680, 681, 689, 690, 692
Graphics.EndContainer( ),680
Graphics.FillRectangle( ),672, 674, 675, 676, 695
Graphics.FromImage( ),695, 896, 897
Graphics.GetHdc( ),704
Graphics.MeasureString( ),688
Graphics.ReleaseHdc( ),704
Graphics.RotateTransform( ),669, 680, 690
Graphics.ScaleTransform( ),666, 669, 680, 690
Graphics.TranslateTransform( ),666, 669, 680
GraphicsPath.AddCurve( ),687
GraphicsPath.AddEllipse( ),677
GraphicsPath.AddLine( ),676, 687
GraphicsPath.CloseFigure( ),687
GraphicsPath.IsVisible( ),687

Guid.NewGuid( ),792

H

Handler.HandleRequest( ),554
HandleRequest( ): in Handler,554
Hashtable.Add( ),389
Highlight( ): in,811
HorizontalTransform( ): in,364
HttpRequest.BeginGetResponse( ),860
HttpRequest.EndGetResponse( ),860
HttpResponse.Write( ),855
HttpWebRequest.GetResponse( ),853
HttpWebResponse.GetResponseStream( ),853

I

ICollection.CopyTo( ),417
IDataAdapter.Fill( ),427, 432, 635, 638, 643, 799, 806
IDataAdapter.Update( ),432, 433, 643
IDataObject.GetData( ),623
IDataObject.GetDataPresent( ),623
IDataReader.Close( ),429
IDataReader.Read( ),429
IDbCommand.ExecuteReader( ),429
IDictionary.Add( ),390, 416
IDictionary.CopyTo( ),416
IDisposable.Dispose( ),461
IEnumerator.MoveNext( ),513
IList.Add( ),414, 513
IList.CopyTo( ),415
IList.GetEnumerator( ),513
Image.FromFile( ),586, 620, 674, 688, 695
Image.Save( ),695
Images.Add( ),598, 620
Increment( ): in Counter,746; in Interlocked,703
IncrementAccess( ): in TwoCounter,731, 743
IndexOf( ): in Array,507, 580; in List,400
Info( ): in Tree,153
Insert( ): in List,400
Int32.ToString( ),743
Interlocked.Decrement( ),704
Interlocked.Increment( ),703
Interrupt( ): in,726, 862
IntHolder.AddFan( ),282
IntHolder.AddPlayer( ),282
Invalidate( ): in BitArray,671; in PictureBox,695, 697
Invoke( ): in,873
IPAddress.Parse( ),829
IsAssignableFrom( ): in,522
IsMatch( ): in,500, 502, 507
IsolatedStorageFile.GetUserStoreForAssembly( ),477
IsSubclassOf( ): in Type,521, 847
IsVisible( ): in GraphicsPath,687
Items.Add( ),609, 613, 620
Items.AddRange( ),610, 613
Items.Contains( ),612

J

Join( ): in Thread,728, 729

K

Key.CompareTo( ),419

L

Label( ): in Forms,566
Label.Focus( ),642
Length.ToString( ),620
Links.Add( ),605
List.Add( ),400
List.Contains( ),400
List.CopyTo( ),400
List.IndexOf( ),400
List.Insert( ),400
List.Remove( ),400
Listeners.Add( ),498
Load( ): in Assembly,541, 846; in XmlDocument,777, 803, 810; in XslTransform,820
LoadFile( ): in,812

M

MakeSundae( ): in Sundae,209
Match( ): in,507; in Regex,540
Matches( ): in,502
Math.Cos( ),678, 684
Math.Floor( ),363
Math.Sin( ),663, 667, 678, 684
Math.Sqrt( ),440
Matrix.Clone( ),683
Measurement.Print( ),177
MeasureString( ): in Graphics,688
MenuItems.Add( ),648, 650, 651, 691, 694, 696, 811
MenuItems.AddRange( ),647, 648
MessageBox.Show( ),605
Model.CloneMenu( ),648

Monitor.Enter( ),736, 741, *769*
Monitor.Exit( ),736, 741
MoveNext( ): in IEnumerator,513; in XPathNodeIterator,803, 805, 809
MoveToNextAttribute( ): in XmlTextReader,776
Mutex.WaitOne( ),752, 753

N

Name.perhapsRelated( ),163
Name.printGivenName( ),163
NameValueCollection.Add( ),393, 419
NameValueCollection.GetValues( ),393
NameValueCollectionEntry.CloneMenu( ),648
Navigate( ): in,700
NewGuid( ): in Guid,792
NewRow( ): in DataTable,422, 423
Next( ): in Random,91, 93, 94, 99, 103, 107, 108, 145, 187, 188, 191, 193, 269, 360, 361, 385, 386, 387, 413, 414, 415, 416, 419, 519, 521, 578, 602, 603, 697, 705, 713, 890, 899
NextDouble( ): in Random,99, 133, 142, 143, 364, 454, 554, 578, 602, 726
Nodes.Add( ),617
Nodes.Clear( ),810
NodeType.ToString( ),808

O

Object.GetType( ),520, 521, 524
Object.Print( ),403
OleDbConnection.Close( ),427, 429, 799, 806, 817
OleDbConnection.Open( ),427, 429, 432, 635, 638, 643, 799, 805, 816
OleDbDataAdapter.Fill( ),816
OnPaint( ): in,662, 666, 669, 672, 674, 676, 687, 688, 690, 693
Open( ): in OleDbConnection,427, 429, 432, 635, 638, 643, 799, 805, 816
OpenFileDialog.ShowDialog( ),651, 695, 697, 812
OpenStandardOutput( ): in Console,799, 801
OpenText( ): in File,827
Out.Close( ),497
Out.WriteLine( ),497

P

Parse( ): in DateTime,829; in IPAddress,829; in SByte,732, 745
PDouble.Wash( ),243, 245
perhapsRelated( ): in Name,163
PictureBox.Invalidate( ),695, 697
play( ): in WriteLine,237
Play( ): in,272, 288
Point( ): in Drawing,566
Pop( ): in Stack,382
Post( ): in,559
Print( ): in Measurement,177; in Object,403
PrintDialog.ShowDialog( ),691
printGivenName( ): in Name,163
println( ): in,240, 241
PrintPreviewDialog.ShowDialog( ),691
Process( ): in Transaction,454
Process.Start( ),605
Push( ): in Stack,382

Q

Queue.Dequeue( ),382
Queue.Enqueue( ),382

R

Random.Next( ),91, 93, 94, 99, 103, 107, 108, 145, 187, 188, 191, 193, 269, 360, 361, 385, 386, 387, 413, 414, 415, 416, 419, 519, 521, 578, 602, 603, 697, 705, 713, 890, 899
Random.NextDouble( ),99, 133, 142, 143, 364, 454, 554, 578, 602, 726
Read( ): in IDataReader,429; in XmlTextReader,775
ReadAll( ): in SourceStream,481
ReadBoolean( ): in BinaryReader,488
ReadByte( ): in BinaryReader,488; in Stream,495
ReadBytes( ): in BinaryReader,488
ReadChar( ): in BinaryReader,488
ReadChars( ): in BinaryReader,488
ReadDecimal( ): in BinaryReader,488
ReadDouble( ): in BinaryReader,488
ReadLine( ): in,466; in StreamReader,831, 834, 836, 839, 841
ReadOnly( ): in ArrayList,385
ReadString( ): in BinaryReader,488
ReadToEnd( ): in StreamReader,793, 853
ReadXml( ): in DataSet,801
RecursiveCall( ): in RecursiveNotLocked,751
RecursiveNotLocked.RecursiveCall( ),751
ReflectModel( ): in,580
Regex.Match( ),540
Regex.Replace( ),503
Regex.Split( ),570
Relations.Add( ),817
ReleaseHdc( ): in Graphics,704
Remove( ): in ArrayList,705; in Controls,572, 705; in List,400

Replace( ): in Regex,503
Resolve( ): in Dns,832, 836, 840
ResourceManager.GetObject( ),593
ResourceSet.GetString( ),588
ResourceWriter.AddResource( ),592
ResourceWriter.Generate( ),592
ResumeLayout( ): in,567
Reverse( ): in Array,357, 476; in ArrayList,414
rollup( ): in Window,235
RotateTransform( ): in Graphics,669, 680, 690
Rows.Add( ),423, 425, 432
Run( ): in Application,562, 567, 572, 576, 580, 582, 583, 584, 585, 586, 588, 593, 595, 597, 600, 603, 605, 609, 610, 612, 614, 616, 617, 621, 624, 632, 635, 638, 643, 649, 652, 661, 663, 667, 671, 673, 674, 675, 676, 677, 684, 687, 689, 690, 691, 693, 695, 697, 700, 705, 713, 732, 745, 812, 897

S

Save( ): in Image,695
SaveFileDialog.ShowDialog( ),651, 695
SByte.Parse( ),732, 745
ScaleTransform( ): in Graphics,666, 669, 680, 690
Screen.GetBounds( ),712
Scrub( ): in,224
Seek( ): in FileStream,495; in Stream,494, 495
SendToBack( ): in Contains,642
Serialize( ): in XmlSerializer,784, 786, 791, 793
Set( ): in BitArray,386, 387
SetDataBinding( ): in DataGrid,635
SetDataObject( ): in Clipboard,623, 631
SetError( ): in Console,497
SetIn( ): in Console,497
SetItemCheckState( ): in,613
SetOut( ): in Console,497
SetPixel( ): in Bitmap,697
SetToolTip( ): in ToolTip,600
Show( ): in MessageBox,605
ShowDialog( ): in ColorDialog,651; in FontDialog,652; in OpenFileDialog,651, 695, 697, 812; in PrintDialog,691; in PrintPreviewDialog,691; in SaveFileDialog,651, 695
Sin( ): in Math,663, 667, 678, 684
Size( ): in Drawing,566, 596, 607, 712
Sleep( ): in Thread,168, 172, 252, 713, 718, 725, 726, 729, 733, 734, 746, 747, 748, 750, 753, 766, 850, 862
SomeLogicalEvent( ): in,569
Sort( ): in Array,357, 360, 361, 417, 420; in ArrayList,414
SortedList.Add( ),391, 392
SortedList.GetByIndex( ),392
SortedList.Synchronized( ),765
SourceStream.ReadAll( ),481
Split( ): in,828; in Regex,570
SplitString( ): in DomainSplitter,571
Sqrt( ): in Math,440
Stack.Pop( ),382
Stack.Push( ),382
Start( ): in,897; in Process,605; in TcpListener,833, 838; in Thread,716, 718, 720, 727, 728, 729, 733, 746, 747, 748, 749, 767, 838, 840
Stop( ): in TcpListener,834, 838
Stream.Close( ),486, 487, 488, 491, 492, 506
Stream.CreateNavigator( ),808
Stream.ReadByte( ),495
Stream.Seek( ),494, 495
StreamReader.Close( ),501, 502, 504, 828
StreamReader.ReadLine( ),831, 834, 836, 839, 841
StreamReader.ReadToEnd( ),793, 853
StreamWriter.Close( ),493
StreamWriter.Flush( ),831
StreamWriter.WriteLine( ),492, 831, 834, 836, 839, 841
String.Format( ),190
StringBuilder.Append( ),831, 843, 861
StringBuilder.ToString( ),831, 861
StringDictionary.ContainsKey( ),396
SubItems.Add( ),620
SuppressFinalize( ): in GC,172
SuspendLayout( ): in,566
Swim( ): in Useful,306
Synchronized( ): in SortedList,765
System.EventHandler( ),566, 568, 574

T

Tables.Add( ),422
TcpClient.Close( ),831, 834, 836, 839, 841
TcpClient.Connect( ),836, 841
TcpClient.GetStream( ),831, 833, 836, 839, 841
TcpListener.AcceptTcpClient( ),833, 838
TcpListener.Start( ),833, 838
TcpListener.Stop( ),834, 838
Thread.Abort( ),733, 746, 747, 767
Thread.Join( ),728, 729
Thread.Sleep( ),168, 172, 252, 713, 718, 725, 726, 729, 733, 734, 746, 747, 748, 750, 753, 766, 850, 862
Thread.Start( ),716, 718, 720, 727, 728, 729, 733, 746, 747, 748, 749, 767, 838, 840
Time( ): in WhatsTheTime,874
ToCharArray( ): in,476
ToolTip.SetToolTip( ),600
ToString( ): in,730, 767; in Counter,188, 190; in DateTime,874; in Int32,743; in Length,620; in NodeType,808; in StringBuilder,831, 861; in WriteLine,836, 840

Trace.WriteLine( ),498
Transaction.Process( ),454
Transform( ): in XslTransform,820
TranslateTransform( ): in Graphics,666, 669, 680
Tree.Info( ),153
TwoCounter.IncrementAccess( ),731, 743
Type.GetConstructor( ),541
Type.GetInterfaces( ),524
Type.GetMethods( ),541
Type.GetType( ),516, 519, 524
Type.IsSubclassOf( ),521, 847

U

UnicodeEncoding.GetBytes( ),481
Update( ): in IDataAdapter,432, 433, 643
Useful.Fight( ),306
Useful.Fly( ),306
Useful.Swim( ),306
UtilityCo.BeginBilling( ),559

V

VerticalTransform( ): in,364

W

WaitOne( ): in Mutex,752, 753
Wash( ): in,244, 249, 250; in PDouble,243, 245
WebRequest.Create( ),853, 860
WhatsTheTime.Time( ),874
Write( ): in BinaryWriter,486, 487; in Console,108, 109, 359, 413, 423, 480, 484, 847; in HttpResponse,855
WriteByte( ): in FileStream,484
WriteLine( ): in Console,72, 77, 78, 83, 89, 90, 91, 93, 94, 95, 97, 98, 101, 102, 103, 104, 105, 108, 109, 117, 118, 129, 131, 132, 133, 134, 135, 136, 137, 142, 143, 145, 150, 151, 153, 154, 155, 157, 162, 163, 165, 168, 169, 171, 172, 174, 177, 180, 181, 182, 184, 185, 186, 187, 188, 191, 205, 207, 210, 211, 220, 221, 222, 223, 224, 226, 227, 228, 229, 230, 231, 232, 240, 242, 243, 244, 245, 249, 250, 252, 256, 262, 263, 264, 268, 271, 272, 273, 275, 276, 277, 279, 280, 282, 284, 288, 289, 290, 291, 293, 295, 303, 304, 306, 352, 353, 356, 357, 359, 361, 364, 369, 382, 383, 385, 386, 387, 388, 389, 390, 392, 393, 395, 396, 397, 398, 406, 408, 409, 410, 413, 414, 415, 416, 417, 420, 423, 426, 429, 432, 433, 445, 446, 447, 449, 454, 457, 458, 459, 460, 461, 462, 463, 464, 466, 474, 475, 477, 480, 485, 488, 492, 495, 496, 500, 502, 503, 506, 512, 515, 516, 520, 521, 524, 529, 530, 531, 534, 535, 536, 539, 540, 541, 548, 549, 552, 554, 555, 558, 559, 560, 574, 608, 609, 610, 611, 612, 614, 617, 625, 626, 628, 642, 651, 662, 666, 687, 695, 697, 700, 703, 704, 716, 717, 718, 719, 720, 725, 726, 728, 729, 748, 749, 750, 751, 752, 753, 766, 775, 776, 777, 789, 793, 799, 803, 805, 809, 810, 825, 828, 831, 833, 834, 836, 838, 839, 840, 841, 842, 843, 844, 846, 847, 850, 853, 860, 861, 862, 874, 898; in Debug,498; in Error,447, 449, 450, 452, 453, 454, 457, 465, 466, 467, 468, 469; in Out,497; in StreamWriter,492, 831, 834, 836, 839, 841; in Trace,498
WriteLine.play( ),237
WriteLine.ToString( ),836, 840
WriteXml( ): in DataSet,799, 816
WriteXmlSchema( ): in DataSet,799, 801

X

XmlDataDocument.CreateNavigator( ),805, 809
XmlDocument.CreateNavigator( ),803
XmlDocument.Load( ),777, 803, 810
XmlSerializer.Deserialize( ),789, 793
XmlSerializer.Serialize( ),784, 786, 791, 793
XmlTextReader.Close( ),776
XmlTextReader.MoveToNextAttribute( ),776
XmlTextReader.Read( ),775
XPathNodeIterator.MoveNext( ),803, 805, 809
XslTransform.Load( ),820
XslTransform.Transform( ),820

Index

Please note that some names will be duplicated in capitalized form. Following C# style, the capitalized names refer to C# classes, while lowercase names refer to a general concept.

A

al.exe (assembly linking utility),195, 203, 204, 266, 385, 591
aliasing,90
array: bounds checking,350
assemblies,532
attributes,540, 760, 824

B

Beck, Kent,887
binding,634, 642
Boehm, Barry,7, 316
bounds checking,350

C

callback,858, 860, 873
CLR (Common Language Runtime),538
comments,83
const keyword,251, 253, 276, 505, 828
constant,51
constants,51
constructor: static,184

D

deadlock,769, 770
default constructor,165, 523, 524
default values,57, 183
delegates: callback functions and,858, 860, 873
dictionary,381

E

Enter( ): in type Monitor,769

F

finally blocks,532
for loops,134, 137

G

graphics,591, 673

H

Hejlsberg, Anders,2, 561

I

IL (intermediate language),77, 78
Integral types,*See* Value types

J

Jones, Capers,7, 9

K

keyword,137

M

McConnell, Steve,7
Monitor.Enter( ),769
multicast delegates,552
multidimensional,190, 357, 435
multidimensional arrays,190

O

operator overloading,278, 279, 282
overloading operators,278, 279, 282

P

patterns: design,865
Picasso, Pablo,3

R

reflection,540

S

Scope of this book,3
sealed classes,254
serialization,52
static constructors,184
Sun,883

T

thread safety,750
Turing, Alan,3

U

upcasting,237, 262, 263, 264
using keyword,660

V

value types,51, 92, 94
Value types: size and default values,57
values, default,57, 183
versioning,52
von Neumann, John,3


Bruce Eckel's
HANDS-ON
JAVA™
SEMINAR
Learn the programming language of the World Wide Web
In this step-by-step introduction each carefully-chosen subject is covered in a lecture followed by hands-on programming exercises.
This course is for you if you can follow basic code examples written in C language syntax.
WWW.BRUCEECKEL.COM